Humboldt-Universität zu Berlin

Masterarbeit

User aiding Web Application for the Generation, Manipulation and Aggregation of RDF Data

Nutzerunterstützende Webanwendung zur Erzeugung, Manipulation und Aggregation von RDF Daten

zur Erlangung des Grades Master of Arts
an der Philosophischen Fakultät I
der Humboldt-Universität zu Berlin

vorgelegt von
Oliver Pohl
(Matrikel-Nr. 534230)

Erstgutachterin: Prof. Vivien Petras, PhD
Zweitgutachter: Alexander Struck, M.A.

Berlin, den 10. November 2014

Abstract

This thesis describes the conceptualization and implementation of *rdfedit*, a web application created with the intent of helping Semantic Web novice users with the generation, manipulation and aggregation of RDF data in a tabular user interface. *rdfedit* connects to the Semantic Web search engine Sindice to provide users with features like the semi-automatic import and remapping of triples from external resources or converting literal objects of triples to appropriate URIs. To examine the usability of *rdfedit* a heuristic usability evaluation as well as a thinking aloud test have been conducted with the result of *rdfedit* fulfilling its goals of enabling Semantic Web newcomers to work with RDF data without needing to have prior knowledge about Semantic Web technologies.

Zusammenfassung

Diese Masterarbeit beschreibt die Konzeptualisierung und Implementierung von *rdfedit*, einer nutzerunterstützenden Webanwendung mit einer tabellarischen Nutzeroberfläche zur erleichterten Erzeugung, Manipulation und Aggregation von RDF-Daten. *rdfedit* nutzt die Semantic-Web-Suchmaschine Sindice, um seinen Nutzern Funktionen zum halbautomatischen Import und Mapping von Tripeln aus externen Quellen sowie zur Konvertierung literaler Objekte zu passenden URIs zu ermöglichen. Zwecks Untersuchung von *rdfedits* Usability (Gebrauchstauglichkeit) wurde eine heuristische Usability-Evaluation und ein Thinking-Aloud-Test durchgeführt. Die Ergebnisse der Untersuchungen zeigen, dass *rdfedit* sein Ziel erfüllt und Semantic-Web-Novizen ohne entsprechende Vorkenntnisse die Arbeit mit RDF-Daten ermöglicht.

Acknowledgements

I would like to express my thanks to everyone who supported me during the implementation of *rdfedit* and while writing this thesis.

At first, I would like to thank Prof. Harald Sack for hosting the "Semantic Web Technologies" MOOC which sparked my interest about this topic. Furthermore, I'd like to thank Dr. Christian Stein and Prof. Michael Seadle for providing deeper knowledge about the Semantic Web and its related technologies.

My thanks also go to Alexander Struck and Konstantin Baierer for hosting the Library Systems course back in 2011. Without that course my deep interest in programming would maybe never have awoken. Additionally I'd like to thank Prof. Vivien Petras for providing me with lots of tasks and challenges during my time of being her student assistant, which helped me develop my academic and programming skills.

I also want to thank the team of the "Digitaler Wissensspeicher" of the Berlin-Brandenburgischen Akademie der Wissenschaften for taking me as an intern and advising me throughout the creation of this thesis: Anett Brüsemeister, Alexander Czmiel, Sascha Grabsch, Marco Jürgens and Josef Willenborg. My gratitude also goes to Svantje Lilienthal for her valuable input.

I would like to thank Christopher Grieser for advising me with the setup and execution of the usability tests described in this thesis. At last, I would like to thank Stefan Pohl, Kati Müller and my pug Goethe for their moral support.

Thank you all, I couldn't have done it without you!

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

AJAX Asynchronous JavaScript and XML

API Application Programming Interface

BBAW Berlin-Brandenburgische Akademie der Wissenschaften – Berlin-Brandenburg Academy of Sciences and Humanities

CSS Cascading Style Sheet

DC Dublin Core

DFG Deutsche Forschungsgemeinschaft – German Research Foundation

DVD Digital Video Disk

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

IP Internet Protocol

ISBN International Serial Book Number

IRI Internationalized Resource Identifier

JS JavaScript

JSON JavaScript Object Notation

JSON-LD JSON Linked Data

MARC Machine Readable Cataloging

Memex Memory Extender

MIME Multipurpose Internet Mail Extension

OAI-ORE Open Archive Initiative - Object Reuse and Exchange

OWL Web Ontology Language

PDF Portable Document Format

RDF Resource Description Framework

RDFa Resource Description Framework in Attributes

REST Representational State Transfer

RSS Really Simple Syndication

SPARQL SPARQL Protocol And RDF Query Language

SQL Structured Query Language

Turtle Terse RDF Triple Language

URI Uniform Resource Identifier

URL Uniform Resource Locator

W3C World Wide Web Consortium

XML eXtensible Markup Language

1 Visions about personal agents

When Vannevar Bush presented his idea of the Memory Extender (Memex) in 1945, he already grasped the basic concepts of today's hypertext. Envisioning exponential growth of scientific information output, he proposes a desk-like machine "in which an individual stores all his books, records and communications and which is mechanized so that it may be consulted with exceeding speed and flexibility." While "consulting" documents quickly, the Memex also uses "associative indexing", which allows users to access referenced records or further relevant information with low effort.

The main implementation of hypertext we use today, the HyperText Markup Language (HTML), was introduced in 1990 by Berners-Lee and Cailliau. HTML documents contain structured information in machine-readable form, that can be interpreted and displayed by web browsers for human users to read. It also enables referencing documents that are scattered throughout the World Wide Web using Uniform Resource Locators (URLs) [Arvidsson et al., 2012], addresses which documents on the Web can be located with, thus creating the "Web 1.0" and realizing Bush's vision of connecting information.

The Memex was intended to be a private device using public information and merging its user's own content with other researchers Memexes [Oren, 1991; Murray, 1993]. This resembles the idea of the Web 2.0 [O'Reilly, 2005] where Internet users start to contribute content and share information with each other. Apart from providing information to its user, Bush's Memex could also be seen as a device to "[encompass] the problem of information overload" and to "control and channel information for use" [Johnston and Webber, 2005].

Furthermore, Oren [1991] extends the Memex and proposes an "adaptive Memex", a device that would act on its own and search, process and display information for its user. In 2001 Internet pioneer Tim Berners-Lee shared his idea of the "Semantic Web" with the world [Berners-Lee et al., 2001]. In his article, Berners-Lee also suggests devices he calls "Semantic Agents" which retrieve and process relevant data, and act on behalf of the desires of its owner. Berners-Lee illustrates an example use case, where the user needs to see a doctor and his Semantic Agent automatically looks up nearby clinics with available time slots and books an appointment.

Apart from being an autonomous assistant, such technology could be used for complex question answering [Antoniou and Harmelen, 2008, p. 4]. A few years ago, it was still very difficult to answer questions like "Which rivers flow into the black sea?"

[d'Aquin et al., 2011] or "How big is the population of Berlin?" without consulting an expert or a dedicated database. The main reasons why technologies like question answering systems and Berners-Lee's Semantic Agent have not been developed to a mainstream extent is that the major amount of data publicly accessible on the "normal web" are only available and indexed in plain text, and the underlying data is commonly not accessible [Domingue et al., 2011]. Now or Microsoft's Cortana can already analyze that information to some extent and answer questions their users ask them on the fly. However, they still fail at understanding and answering more complex questions.

Only being fed plain text information, a machine cannot distinguish whether you meant "Berlin, Germany" or "Berlin, New York" when you queried for Berlin's population size. Current smartphone software products like Apple's Siri, Google

Semantic Web technologies enable a machine-interpretable disambiguation of information by attaching relationships between resources and either storing those relationships in a database or marking up that information in HTML documents. Since those relationships ought to be meaningful, i.e. they should signify a meaning, such information is *semantic* (Greek sēmantikós (σημαντικός), significance, see Lidell and Scott [1940]). Using that technology, you can interlink data in a more meaningful way and thereby facilitate the answering of complex questions, inferring new knowledge and paving the way for personal semantic agents with intelligent appointment scheduling.

This section (1) introduced the idea connecting information semantically and presents an outline for the rest of this thesis. Section (2) introduces the concept of the Semantic Web and the Resource Description Framework (RDF). So far, the adoption of Semantic Web technologies only advance slowly because it appears to be too complex for the normal Internet user. Section 3 presents the idea of *rdfedit*, a web application that helps Semantic Web newcomers with the creation of RDF data – data for the Semantic Web. Within that section, the goals and key features that *rdfedit* should provide are defined and the market of Semantic Web related software is being investigated. *rdfedit*'s technical implementation, structure and workflows of each key functionality are described thoroughly in section 4.

To test whether *rdfedit* can actually aid Semantic Web novices with the generation of RDF data, a heuristic usability evaluation as well as a thinking aloud test with four participants have been performed. Section 5 discusses the setup and outcomes of those usability evaluation methods and additionally contains some retrospective self-criticism on what can still be improved and implemented regarding *rdfedit*. This thesis closes

with section 6, concluding what has to be done next to increase the popularity and accessibility of the Semantic Web.

2 Semantic Web Technologies

2.1 Making Data Explicit

In order to reach the visions described in section section 1, we need to make use of technologies that allow us to apply semantics to digital information while utilizing common web standards. The major part of the visible web is being created mainly by and for humans using HTML. While we as humans are able to interpret the meaning of what our Internet browser displays, our computers cannot. We are accustomed to give implicit information in documents a certain meaning, such as a big, bold font at the top of a page can be interpreted as a heading [Blumauer and Pellegrini, 2006]. To make that information comprehensible for machines, we have to make the implicit information only we as humans understand explicit for the computers.

Regarding documents, many of them contain implicit and explicit metadata (data about data). One example for implicit metadata are big, bold words on top of a page being a title or chapter heading of a document. Respectively, explicit metadata are when some data about that document itself, what the title or who the author is, exists in a machine readable manner. Regardless of explicity or implicity, metadata add some meaning to the document they describe [Antoniou and Harmelen, 2008].

Using explicit metadata, we can describe all things of the same type using the same method. For instance, a book always has an author, a title and an International Serial Book Number (ISBN), thus we assign metadata fields for each of those respective values. The same applies to cars, which always have a manufacturer, a color and a year of manufacture. In other words, all books (or cars) have something in common: their characteristics. If you aggregate all characteristics (henceforth attributes) of a certain entity, you define a *class* about it. A class functions as a template which allows you to describe every possible variation (instance) of that entity. So, using author, title and ISBN you create a book-class that serves as a template for describing (instantiating) books. For example, the book *Moby Dick* by *Herman Melville* with the ISBN *978-3800054794* is an instance of the book class. When we instantiate a special book, e.g. *Moby Dick* we inherently state, that *Moby Dick* is a book, since we have applied the book-class. We call the resulting metadata-record a *resource*.

While the book *Moby Dick* belongs to a class (the book-class), its author does too. An author is a person having a name, birthday and birthplace. So when we examine the example of *Moby Dick* closely, we find *relationships* between two resources: `The person ‘‘Herman Melville’’ – is the author of – the book ‘‘Moby Dick’’`.

Summarizing, we group similar *resources* together to *classes* and establish *relationships* between *resources*.

2.2 Applying Semantics to the Web

On the Semantic Web, we can store classes, resources and their relationships among another and access them using URLs [Arvidsson et al., 2012]. Since one of the core ideas of the Semantic Web is to be unambiguous [Berners-Lee et al., 2001], the utilization of Uniform Resource Identifiers (URIs) [Masinter et al., 2005] is preferred, because every URI is unique. Together with the HyperText Transfer Protocol (HTTP), which allows communication and data exchange between computers, URIs shape the foundation of Berners-Lee's "semantic web layer cake" (see Figure 2.1). It illustrates which technologies and standards are used to form the semantic web and how they build and on one another.

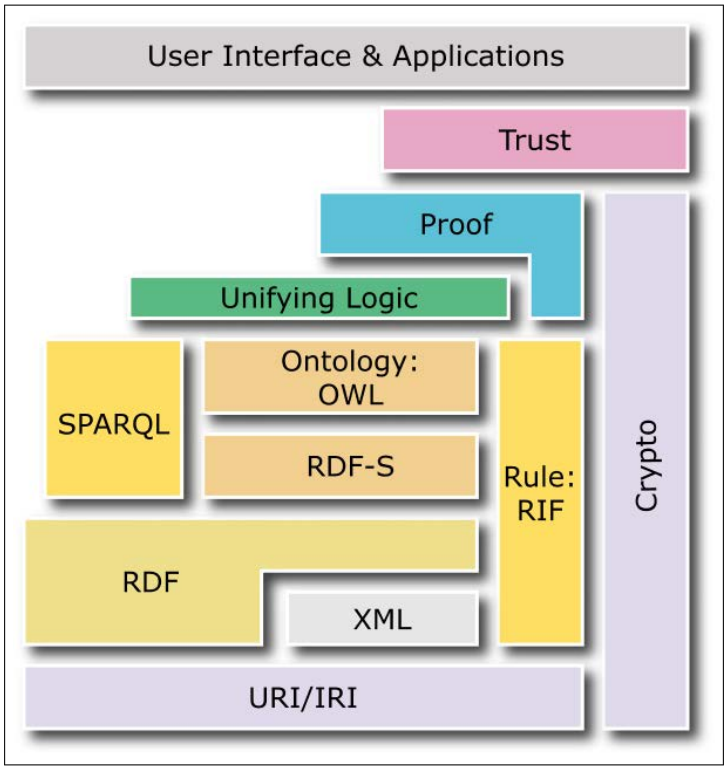

Figure 2.1: Latest version of the semantic web layer cake, taken from Bratt [2007]

To express the semantic relations, we use the RDF [Lassila and Swick, 1999]. The basic building blocks of RDF consist of three elements:

- A *Subject*: a URI-identified resource that is being described,
- a *Predicate*: a URI-identified reused specification of the relationship, and
- an *Object*: a resource (URI) or literal to which the subject is related.[1]

Together, subject, predicate and object form a *triple*. Those triples can be expressed in various syntaxes. The most commonly used expression for RDF data exchange is RDF/XML [Gandon and Schreiber, 2014] (see Figure 2.1, enclosed by RDF). In this thesis, Turtle (Terse RDF Triple Language) [Beckett and Berners-Lee, 2008] will be used to express RDF examples. Taking the Moby Dick example from section 2.1, we can state the following facts using natural language and Terse RDF Triple Language (Turtle):

Natural Language	**Turtle**		
	Namespace Declaration		

	`@prefix ex: <www.example.org/ns#> .`
	`@prefix bibo: <http://purl.org/ontology/bibo/> .`
	`@prefix dc: <http://purl.org/dc/elements/1.1/> .`		

	Subject	**Predicate**	**Object**
Moby Dick is a *book*.	`ex:mobyDick`	`a`	`bibo:book.`
Moby Dick has the title "Moby Dick".	`ex:mobyDick`	`dc:title`	`''Moby Dick''.`
Moby Dick has the ISBN "978-3800054794".	`ex:mobyDick`	`bibo:isbn`	`''978-3800054794''.`
Moby Dick was authored by *Herman Melville*.	`ex:mobyDick`	`dc:creator`	`ex:hermanMelville.`
Herman Melville is an *author*.	`ex:hermanMelville`	`a`	`ex:author.`

Table 2.1: A short example describing the book "Moby Dick" in natural language and Turtle

The example in Table 2.1 makes use of different namespaces for describing the book *Moby Dick*. Behind those namespaces lie ontologies and vocabularies can be used to create meaningful statements. Studer et al. [1998] describes an ontology as "a formal explicit specification of a shared conceptualization of a domain of interest", i.e. the ontology predefines how one should understand the relations and classes, when we apply them. For example, `dc:creator` resolves to the creator-member of the Dublin Core (DC)

[1] Adapted from: Norton [2013]

vocabulary[2], where it is defined as "an entity primarily responsible for making the resource". In our particular example, we thereby state that the resource *Moby Dick* was created by the resource *Herman Melville*. By creating this relation between those two resources, we interlink them.

Using this principle of creating and interlinking many and diverse resources, we generate Linked Data. If you cluster Linked Data records by their location and visualize the relationships between those clusters, you generate the Linked Open Data cloud [Bizer et al., 2009].

This passage only provided an abridged and simplified version of the basic principles of Linked Data, without going into further technical details. Still, for people new to the topic, Linked Data and the Semantic Web might be hard to grasp, which might explain the slow adoption of these technologies.

2.3 The Notion of the Semantic Web

Nixon et al. [2011] expect the Semantic Web to have taken roots as a mainstream technology by 2019. For that to happen, we have to face one major challenge in order to establish the semantic web as such: *Make Semantic Web technologies comprehensible*.

Most people working with (meta-) data do not always have a technological background, making it hard for them to adopt a necessary skill set for creating, using and querying RDF data [Salo, 2013]. Thus, Semantic Web technologies are either "hard to explain" [Benjamins et al., 2011] or hard to understand.

Moreover, you need to use those technologies to create more than just metadata, i.e. create use cases for the average Internet user, to advance Linked Data and related technologies to common use [Stuart, 2011, p. 38]. Such use cases mainly involve enriching and exposing your data within HTML code. Using the Resource Description Framework in Attributes (RDFa) [Adida et al., 2012] you can add semantic annotations to HTML source code and thereby make the statements and relations accessible and interpretable for web crawlers, search engines and other services. Hence you create additional value to your data.

The distribution of RDFa among public websites is on the rise. When Mika and Potter examined websites in the Bing[3] corpus for the utilization RDFa in 2012, they discovered around 4.7 percent of those websites made use of RDFa. One year later, Bizer et al.

[2] `http://dublincore.org/documents/2012/06/14/dcmi-terms/?v=terms#creator` — This URL as well as all other URLs in this thesis have last been tested on November 9th, 2014.

[3] `www.bing.com`

[2013] conducted a similar examination, this time scanning the websites contained in the Common Crawl Index[4] for RDFa use, obtaining a slightly higher result of 5.64 per cent (169 million unique URLs). When analyzing the Linked Open Data Cloud as a whole, the growth between 2011 and 2014 becomes eminent. The recent study of Schmachtenberg et al. [2014a] showed that the Linked Open Data cloud grew by 271 per cent regarding data sets when compared to the foregone analysis by Jentzsch et al. [2011] (from around 300 to over 1000 major data sets).

While these numbers indicate a rising adoption and application rate of Semantic Web technologies in general, they still seem to be a niche product. Sletten [2014] presumes the reason for Semantic Web technologies not picking up a faster pace are businesses. They cannot integrate those technologies in a useful way or fail to see the innovative potential the Semantic Web might bring. Breslin et al. [2010] argues that although there is a great potential adopting Semantic Web technologies, but only if they are incorporated across all participants in the industry, i. e. enabling interoperability between businesses and customers by agreeing on a few technological standards and giving up their own self-crafted or bought business solutions.

These assumptions are being confirmed by Cardoso [2007], who surveyed the usage behavior of Semantic Web technologies in the United States. His results state that more than 80 per cent of all participating Semantic Web users are involved in academia or academia-industry collaborations, rather than working with Semantic Web technologies in the industry. Simultaneously, the adoption rate of Semantic Web technologies in general rises, as Janev and Vranes [2009] determined a few years ago.

This indicates that the user base of Semantic Web technologies is on the edge from the innovators phase to the early adopter stage when grouping the Semantic Web community into Rogers' (2003) diffusion model of innovations.

To raise the adoption rate of Semantic Web technologies, we need to provide software that supports or automates the creation of RDFa markup. Hendler [2001] thinks that most users "should not even know that Web semantics exist" and "semantic markup should be a by-product of normal computer use". For example, Corlosquet et al. [2009a] created a semantic web technologies toolkit for the popular web content management software Drupal[5] - `Drupal RDFCCK` - which created semantic markup while the user created new web content. Additionally, you do not even need to know very much about Drupal to install and configure that toolkit [Corlosquet et al., 2009b], hence this soft-

[4] `http://commoncrawl.org/common-crawl-url-index/`
[5] `https://drupal.org/`

ware complies with Hendler's ideal of making the creation of semantic markup invisible for the user. In a later version of Drupal, the RDFCCK toolkit was officially integrated, thus enabling every Drupal user to create semantic markup [Havlik, 2011].

Another approach for cultivating the Semantic Web is to provide tools that emulate the behavior of software users are already acquainted with. For example, the relational data base model [Codd, 1970] has been well established throughout the past decades with the rise of relational databases and corresponding query languages such as Structured Query Language (SQL). Loosely, you can imagine a triple store (RDF database) also having tables as in their SQL counterparts, but when it comes to RDF, the columns in those tables become more flexible [Newman, 2007]. While the relation between data in SQL is defined by their columns, in RDF relation is inherently explained in each triple. For instance, if you wanted to describe books in SQL, you would have to create a column for each field (e.g. author, title, ISBN) and each row in that table would represent a book.

Inherently, SQL itself is not suitable for the Semantic Web, since you can only describe things you have created columns for. This makes the dynamic description of other data than previously anticipated, e.g. the birth date and birthplace of a book's author in our SQL example, impossible. Moreover, most SQL data bases are just contained within themselves and lack of communication to other, external data bases. When being queried, an SQL data base assumes that only its own data is relevant for the query, and if there are no results then nothing is relevant, hence representing a Closed World Assumption approach [Reiter, 1978].

On the contrary, RDF and triple stores adhere to the Open World Assumption. This means, in a scenario of returning no results to a query, the underlying data base admits that it just does not *know* any results instead of simply stating that no information fitting the query exists [Smets, 1990]. RDF was designed to overcome the flaws of static relational data bases by making it possible to establish links between vocabularies and data bases at different locations [Magee, 2011]. When obtaining an empty result list from a triple store, it means you might get the information you need elsewhere when traversing the Linked Open Data cloud. Because of their interoperability RDF and Semantic Web technologies perform better on complex tasks such as enriching local data with external content and deriving new information from the already existing knowledge base [Bergmann, 2009].

Having the advantages and problems in mind, this thesis describes the creation and evaluation of a user-supporting web application for the creation, manipulation and ag-

gregation of RDF data. This application *rdfedit*[6] follows Newman's [2007] proposal of providing the users with features they are already acquainted with and delivers a tabular user interface for the interaction with RDF data. The following chapter discusses the requirements that *rdfedit* should match in order for it to function as a useful and purposeful application in the Semantic Web cosmos.

[6] Source code repository available at: `https://github.com/suchmaske/rdfedit`

3 rdfedit

The beginning of this chapter defines the goals *rdfedit* tries to achieve followed by a description of requirements and features on how the set out goals can be most efficiently reached. With the potential features in mind this chapter concludes with a comparison and differentiation of *rdfedit* with other tools being used in a Semantic Web related context in order to position *rdfedit* on the vast map of Semantic Web applications.

3.1 Goals

The idea for creating *rdfedit* originated during an internship at the DFG funded project "Digital Knowledge Store"[7] located at the Berlin-Brandenburgische Akademie der Wissenschaften (BBAW) in 2013. The task of the "Digital Knowledge Store" is to process data generated in the various ongoing BBAW projects, convert them to RDF and store them in a triple store (Semantic Web database). Since that data is very heterogeneous to be batch-processed by a conversion tool, the idea arose to create a tool so members of the BBAW could easily create valid RDF data themselves. The finished product should then be integrated into the "Digital Knowledge Store" environment.

The main goal of *rdfedit* is to enable users to create and manipulate RDF data in a short period of time, maintaining a good data quality while the users do not need to know much about Semantic Web technologies. Hence, the main research question of this thesis is whether *rdfedit* can live up to that goal and if not, to determine the issues that is preventing *rdfedit* from reaching that goal.

rdfedit seeks out to reach people with little to no prior experiences regarding the Semantic Web and make them able to create RDF instance data. One potential audience of *rdfedit* are people who work in cultural heritage institutions and want to create metadata (i.e. instance data) for the objects in their institution's collection. Hence, the main application of *rdfedit* is to help create such data and make the resulting data sets compatible with the Semantic Web.

Since the majority of the users are expected to have no Semantic Web proficiency and might also not be following the latest trends in technology, the target audience of *rdfedit* can be labeled as "novice users" according to Nielsen's [1994, p. 44] user cube. In contrast there are "expert users" who posses the skill to install and configure complex computer applications, model metadata schemata or ontologies and have already extensive knowledge regarding the Semantic Web and Linked Open Data.

[7] `http://wsp.bbaw.de/`

When installing *rdfedit*, "expert users" should configure *rdfedit* in a way it can be easily accessed by the "novice users" and the resulting data complies with the metadata schemata or ontologies of their institution's preference. Hence, the responsibility of obtaining good quality RDF data is split among three parties:

(a) *rdfedit* should provide users with features for the creation and manipulation of RDF data. It also should allow users to import data from external resources for data reuse purposes.

(b) *Novice Users* are responsible for creating RDF instance data and inspect the data they have created and imported for errors.

(c) *Expert Users* can configure how the RDF import functions should be executed. By that they can determine where to fetch data from and which parts of that data should be imported, so they can make sure the data created by the novices users conform with their underlying ontology or metadata schema.

In general, *rdfedit* should adhere to the Linked Data principles proposed by Berners-Lee [2006], such as reusing already existing vocabularies and linking to resources. One key feature of *rdfedit* should be the aggregation of RDF triples from external resources. When reusing that data, the user automatically attaches links from the graph she is currently editing to the graph the triples are imported from, thus increasing the number of bonds within the Linked Open Data Cloud [Grimmes et al., 2012].

Moreover, the RDF data generated with the help of *rdfedit* should also conform to the Linked Data Principles of Heath and Bizer [2011, p. 26]. These imply having users create URIs to enable standardized access mechanisms like HTTP and thereby make hyperlinked-based data discovery possible. Moreover their data should be self-descriptive, allowing machines and humans to interpret their data without difficulty.

To encourage the application of the aforementioned principles, *rdfedit* has to provide functionalities to support its users in using appropriate vocabularies, choosing the correct URIs and creating a valid and consistent RDF graph.

3.2 Requirements & Intended Features

In order to fulfill the goals intended for *rdfedit*, some key features need to be developed. Table 3.1 lists these goals and proposes solutions on how those goals can be met. The subsequent paragraphs explain the solutions in more detail.

Goal	Solution
Valid & Consistent Graph	Auto-completion, Bulk Editing
Reuse Data	Auto-completion, Triple Import & Mapping
Use URIs	Triple Import, Literal-to-URI-Conversion

Table 3.1: Goals of rdfedit *and features to achieve those goals*

3.2.1 Auto-completion

As Schmachtenberg et al. [2014a] have shown, the majority of data sets in the Linked Open Data Cloud make heavy use of only a few vocabularies. The `rdf` vocabulary is being used in more than 98 per cent of all data sets, whereas `rdfs`, `foaf` and `dcterms` appear in more than half of all data sets available in the Linked Open Data cloud. In general Schmachtenberg et al. determined 18 vocabularies that appeared at least 5 per cent of all data sets. Throughout their analysis, they encountered over 600 vocabularies, showing that only a small fraction of available vocabularies are well known and applied widely.

Due to that fact, *rdfedit* should incorporate these major vocabularies and use them to suggest users appropriate predicates when they want to add new triples. These suggestion can either happen adaptively, i.e. rdfedit notices a certain predicate could fit, or via auto-completion. Using the latter, users can start typing a predicate they want to use, although they might not now the exact one, and *rdfedit* filters all preloaded vocabulary entries and presents the user with matching ones. For example, a user wants to add a statement about someone having a name. Hence he types `name` and is being offered `foaf:name`.

3.2.2 Bulk Editing

There exist multiple serializations of RDF, such as RDF/XML [Gandon and Schreiber, 2014], Turtle and RDF/JSON [Davis et al., 2013], all being able to express the same triples in a different way [Manola et al., 2014]. They all represent hierarchical structured data, with RDF subjects being at the top-level, moving to predicates at the mid-level and objects at the bottom-level.

Subject	Predicate	Object
ex:mobyDick	a	bibo:book
ex:mobyDick	dc:title	"Moby Dick"
ex:mobyDick	dc:creator	ex:hermanMelville

Table 3.2: The triples expressed in Listing 3.1 as a triple-table, as intended for rdfedit *(namespace declaration omitted).*

Using Turtle as an example serialization, one could express multiple RDF triples about one subject with only using that subject once (see Listing 3.1). Since the target audience of *rdfedit* probably does not know how to use any of these serializations, the triples should be broken into a simple subject-predicate-object-table, where every triple is presented by a single row, as shown in Table 3.2. This implies each row having a subject-URI that can appear multiple times in the subject-column.

When users alter a single subject-URI in the RDF-table, they also alter the structure of the RDF graph: The affected triple is being removed from the set of triples with the same subject-URI, thus removing information about the resource behind that URI. Depending on the predicate-object-tuple of those triples, errors in the data schema could arise. For example, when describing an OAI-ORE [Lagoze et al., 2008] resource map using RDF it is important to state, which resource is being described. Omitting that crucial information would not only make the resulting file invalid for the OAI-ORE schema but it would also make less sense for humans, since we could not tell anymore what we created that resource map for in the first place.

```
@prefix ex: <www.example.org/ns\#> .
@prefix bibo: <http://purl.org/ontology/bibo/> .
@prefix dc: <http://purl.org/dc/elements/1.1/> .

ex:mobyDick a bibo:book;
    dc:title "Moby Dick";
    dc:creator ex:hermanMelville .
```

Listing 3.1: Example RDF records expressed in Turtle using a subject only once

To avoid such inconsistencies, users should be able to apply the changes made to a subject-URI of one triple to all triples with the same subject in the graph.

3.2.3 Triple Import & Mapping

Schmachtenberg et al. [2014a] recently published the current version of the Linked Open Data cloud, including almost 1100 Linked Data data sets containing around 900,000 documents that describe more than 8 million resources in 188 million triples [Schmachtenberg et al., 2014b], although the actual number of triples inside the Semantic Web is estimated to be greater than 31 billion [Jentzsch et al., 2011].

Depending on how good your discipline is already represented in the Linked Open Data cloud, there might be a chance that someone already created RDF statements about a resource you want to describe. To encourage reuse of data as proposed by Berners-Lee [2006], users of *rdfedit* should be able to somehow look up triples that might be relevant for the task they want to carry out. For instance, a researcher in the field of film studies wants to create a RDF graph describing his or her collection about the movie actor *Wil Wheaton*. Apart from the all-purpose triple store DBPedia[8], there exist other knowledge bases that revolve around movies and television, such as the Linked Movie Database[9] or the EUScreen project[10].

Instead of having to enter every triple describing the collection manually, the researcher could just send a query to a centralized database that has indexed the majority of the Linked Open Data cloud. That service then looks for RDF data (graphs and triples) which suit the researcher's needs. The researcher can pick the data that appears most relevant and thereby import that data into the local graph. Administrators of a *rdfedit* instance can configure the parameters on how those queries should be executed.

Continuing the example, that look-up service would return graph URIs about *Wil Wheaton* coming from the DBpedia and from the Linked Movie Database, containing information in which movies and TV shows Mr. Wheaton starred in. The user selects a graph URI such as `dbpedia:Wil_Wheaton`[11] and *rdfedit* imports information like Mr. Wheaton starred in Star Trek, The Big Bang Theory and is hosting a webshow called "Tabletop".

Moreover, *rdfedit* automatically maps the imported data to a schema the user prefers. However, you cannot presume that everyone has detailed knowledge about metadata or even various metadata schemata. Experts in that domain can configure, how data from external graphs should be mapped into *rdfedit*-users' local graphs. In our case, that administrator defined that triples of originating from the DBpedia should be mapped ac-

[8] `http://dbpedia.org/About`
[9] `http://linkedmdb.org/`
[10] `http://lod.euscreen.eu/`
[11] Full URI accessible in a browser: `http://dbpedia.org/page/Wil_Wheaton`

Knowledge Base	Original Predicate	Mapped Predicate
DBPedia	dpedia:host	dc:contributor
	dbpedia:starring	dc:contributor

Table 3.3: Example mapping for triples about actors in the DBPedia

cording to Table 3.3. Here, the predicates dbpedia:host and dbpedia:starring would be mapped to the more common term dc:contributor. Table 3.4 illustrates that mapping process using some triples found in the graph of dbpedia:Wil_Wheaton.

Subject	Predicate	Object
External Graph: dbpedia:Wil_Wheaton		
dbpedia:Tabletop_(Web_series)	dbpedia:host	dbpedia:Wil_Wheaton
dbpedia:Star_Trek:_The_Next_Generation	dbpedia:starring	dbpedia:Wil_Wheaton
Local Graph: ex:Wil_Wheaton		
ex:Tabletop	dc:contributor	ex:Wil_Wheaton
ex:Star_Trek:_The_Next_Generation	dc:contributor	ex:Wil_Wheaton

Table 3.4: Applied mapping using the mapping configuration described in Table 3.3 (namespace declaration omitted)

To make the import of triples from external graphs possible, there has to be an underlying mechanism that offers a fast look-up service for RDF graphs and triples using keyword queries. Suitable solutions and products will be discussed later throughout this thesis in section 4.3.11 (p. 39).

3.2.4 Literal-to-URI-Conversion

In RDF the object part of triples can take two forms: a URI or a literal. The latter is a tuple consisting of a literal value and an Internationalized Resource Identifier (IRI) that denotes the type of that literal, such as an integer number, a date or a string [Cyganiak et al., 2014]. Compared to URI-objects, literal objects are not dereferenceable, meaning they don't point to any other resource in the Web.

For Semantic Web newcomers, the concept of applying URIs is rather new. Historically, literals in combination with some kind of encoding have been used to create metadata. When adding metadata (outside a Semantic Web context) about *Moby Dick* these literals could take a simple form as: *Author: Herman Melville, ISBN: 978-*

3800054794; or when using Machine Readable Cataloging (MARC)[12] the same statements could have been expressed as: *100 1#$aMellville,Herman,$d1819-1898* and *020 ##$a9783800054794*.

Using the encodings correctly always takes effort for the people applying them. Although the application of literals is sometimes inevitable, the utilization of URIs is being demanded more thoroughly [Berners-Lee, 2006; Grimmes et al., 2012]. Creating triples in RDF is also just applying an encoding during data creation. Assuming that people are accustomed to the concept of a simple key-value principle (*Key: Value*), *rdfedit* can convert these user-given literal-values to appropriate URIs.

For example, users are required to make statements about what location a resource depicts using Geonames-URIs, such as: *This postcard shows the city of Berlin, Germany*. Unfortunately, not all URIs are as self-descriptive as the ones coming from the DBPedia. Whereas Berlin, Germany has an easy to remember or easy to self-construct URI: `dbpedia:Berlin`, Geonames.org does not. There, the URI is: `http://sws.geonames.org/2950159/about.rdf`. Using *rdfedit*, a user could just enter "Berlin" as a literal object, the application seeks the the appropriate URI and uses it to replace the literal. Table 3.5 demonstrates that substitution process using the triple-table view as intended for *rdfedit*.

Subject	**Predicate**	**Object**
Before Substitution: Literal Object		
`ex:Postcard123`	`dcterms:spatial`	`''Berlin, Germany''^^xsd:String`
After Substitution: URI Object		
`ex:Postcard123`	`dcterms:spatial`	`http://sws.geonames.org/2950159/about.rdf`

Table 3.5: An example illustrating the literal-to-URI-conversion process intended for rdfedit *(namespace declaration omitted)*

As mentioned, the literal-to-URI-conversion and triple import features require an external service. The subsequent paragraphs elaborate on suitable solutions as well as they discuss applications and services that try to make the Semantic Web more accessible, comparing them with the the intended features of *rdfedit*.

[12] `http://www.loc.gov/marc/`

3.3 Semantic Web Applications

3.3.1 Creating & Editing Data

In his survey about user preferences regarding Semantic Web technologies, Cardoso [2007] determined that the ontology editor *Protégé*[13] dominates the market of RDF and ontology editors. Since it has already been released in the late 1980s as a tool for knowledge acquisition [Musen, 1989], and later on integrating Semantic Web capabilities [Gennari et al., 2003; Knublauch et al., 2004], *Protégé* had a head start to become established as the ontology and Semantic Web editor of choice. Through the years, *Protégé* has become a very powerful and complex tool using graph visualization and allowing the implementation of external plugins for further features to support its user-base in the creation of OWL [Group, 2012] ontologies and RDF graphs.

Another major ontology editor with plugin capabilities is the *NeOn Toolkit*[14]. Erdmann and Waterfeld [2012] describe its user interface as "accessible to users that do not have long experience with ontologies [...]". Haase et al. [2008] had a similar idea to *rdfedit's* triple import features: They developed a plugin for the *NeOn Toolkit* that allowed ontology engineers to query the Semantic Web search engine Watson[15] and import relevant data into their ontology for information reuse purposes. As for now it seems, that the development of the *NeOn Toolkit* has been halted.[16]

Similar to the idea of *rdfedit*, Lilienthal [2014] created a web application called Triple Geany that allowed Semantic Web novices to create triples by using predefined forms.

While *Protégé* and *NeOn Toolkit* focus on ontology engineering, *rdfedit* as well as Triple Geany follow a simpler approach: the creation of RDF instance data. Experts can make use of such ontology editors and create ontologies, while Semantic Web novices can create data using *rdfedit* that conforms with these ontologies.

3.3.2 Storing, Wrapping & Editing Data

The common way to store RDF data efficiently, connect them to the Linked Open Data cloud and make them able to query is to index them using triple stores — databases for

[13] `http://protege.stanford.edu/`

[14] `http://www.neon-project.org/nw/Welcome_to_the_NeOn_Project`

[15] `http://watson.kmi.open.ac.uk/WatsonWUI/`

[16] At the time publishing this thesis, the download of the *NeOn Toolkit* and further related websites were not accessible. Furthermore, there haven't been any news about this software since 2012.

RDF. Triple stores such as *OWLIM*[17] offer further benefits, such as forward-chaining: When all triples have been inserted into the database, further RDF statements are being inferred from the already existing ones using dedicated reasoner software [Kiryakov et al., 2005]. When the indexing process is finished, the data can be queried using the SPARQL Protocol And RDF Query Language (SPARQL) [Prud'hommeaux and Seaborne, 2008].

The insertion of new data is done by uploading RDF triples directly to the triple store. Once in there, updating or deleting single triples becomes difficult. Before the newer version of the SPARQL standard (1.1) was introduced in 2013, it was not possible to perform updates or deletions on single triples inside a triple store [Arnada et al., 2013]. The data containing the alterations had to be either re-indexed, or other solutions like SPARQL/Update by Hewlett-Packard [Seaborne et al., 2008] had to be taken into consideration.

There are more user-friendly solutions when it comes to editing data in a triple store. For example, *OntoWiki*[18] connects to a triple store and makes the data easier mutable by presenting a collaborative Wiki Environment [Auer et al., 2006]. In a similar fashion, DBTropes[19] - the Semantic Web representation of TVtropes[20] - also offers a user interface where users can collaboratively edit data and directly influence the triple store [Kiesel and Grimnes, 2010].

These dynamic methods to edit RDF data are also an advancement for the people administering the triple store. A few years ago, the DBpedia did not receive immediate updates when there was information added or changed in the Wikipedia. Instead, a RDF data dump had to be generated from the Wikipedia for the DBpedia on a regular basis [Bizer et al., 2009]. Later on, methods for the live extraction of new or altered information had been implemented which reflect the changes on the Wikipedia faster onto the DBPedia [Morsey et al., 2012].

While these solutions offer a user interface overlay for editing data in a triple store, *rdfedit* should be kept separate from databases. When being done creating and editing RDF data with *rdfedit*, users should download a file. Since the user base of *rdfedit* is intended to consist of Semantic Web newcomers, the data generated will probably be forwarded to Semantic Web experts who can cross-check the data and upload them to their triple store.

[17] `http://www.ontotext.com/owlim`
[18] `http://aksw.org/Projects/OntoWiki.html`
[19] `http://skipforward.opendfki.de/wiki/DBTropes`
[20] `http://tvtropes.org/`

3.3.3 Searching Data

The standardized way to retrieve triples is by submitting a SPARQL query to a triple store, similar to using SQL in relational database management systems. Using SPARQL does not only require knowledge about the query language itself, but also about Linked Data vocabularies and the data inside the triple store. Hence, Semantic Web newcomers might fail submitting valid queries.

To make it easier to search the Semantic Web, researchers have developed approaches on how to search triple stores using natural language. In a small study, Kaufmann and Bernstein [2007] have determined, that users prefer the use of interfaces that provide natural language query processing when searching the Semantic Web.

For example, Shekarpour et al. [2013a] presented a method on how to construct SPARQL queries from templates and natural language query input by users (see also Shekarpour et al. [2013b]). There are also approaches that focus on the exact opposite: For users who have to use SPARQL queries but are having a hard time interpreting them, Ngonga Ngomo et al. [2013] have provided methods to convert SPARQL queries to natural language.

Nevertheless, it might be the case that users want to send a query across multiple triple stores or databases they don't even know exist. Hartig et al. [2009] developed a method to execute SPARQL queries over multiple triple stores. Still, executing such a query would require a deeper knowledge about SPARQL and the Semantic Web.

Semantic Web search engines such as Sindice[21] [Oren et al., 2008], Swoogle[22] [Ding et al., 2004], or Watson[23] [d'Aquin et al., 2007] all offer a search interface that resembles the popular search engine Google, hence offering a search interface that can easily be understood by everybody. Gottron et al. [2012] remarked, that these service actually lack the "Google feeling" since they do not offer any query refinement or try to determine related queries.

Sindice and Watson both offer Application Programming Interfaces (APIs), allowing external applications to use their services. Semantic Web browsers, questions answering tools or plugins for the *NeOn Toolkit* make use of Watson's API to extend their functionality [d'Aquin et al., 2008]. The main difference between Sindice and Watson is their depth of information retrieval. While Watson can look up RDF graphs at different locations and their contents, Sindice only provides the look-up service but posesses the

[21] `http://sindice.com/`
[22] `http://swoogle.umbc.edu/`
[23] `http://watson.kmi.open.ac.uk/WatsonWUI/`

greater index. Sindice locates relevant RDF graphs, so users can utilize and process them for their purposes. Sindice's result list interface doesn't always clarify why a particular result is relevant, although that kind of behavior is being expected from search engines nowadays [Tombros and Sanderson, 1998].

When it comes to the features intended for *rdfedit*, like importing triples from external resources, Semantic Web search engines and their APIs seem to be more suitable than SPARQL queries. The search engines have already indexed large amounts of RDF data so it can be queried and retrieved quickly. It is sufficient to send keywords to the search engines to obtain results from multiple resources. Executing SPARQL queries on the other hand would require a deeper understanding of the data structure in the triple store that is to be queried. Moreover, retrieving results from multiple triple stores at once can be very difficult. Since the payoff regarding time and easiness is greater, *rdfedit* should rely on Semantic Web search engines rather than SPARQL. Section 4.3.11 (p. 42) discusses, which search engine does fulfill the needs of *rdfedit* best.

3.3.4 Editing with *rdfedit*

Having looked at some software solutions that offer RDF editing capabilities, it becomes emergent that *rdfedit* should be something simpler than the presented options. First, *rdfedit* does not try to fulfill ontology engineering needs such as *Protégé* or the *NeOn Toolkit*, neither is it intended to make direct changes in a triple store like *OntoWiki*.

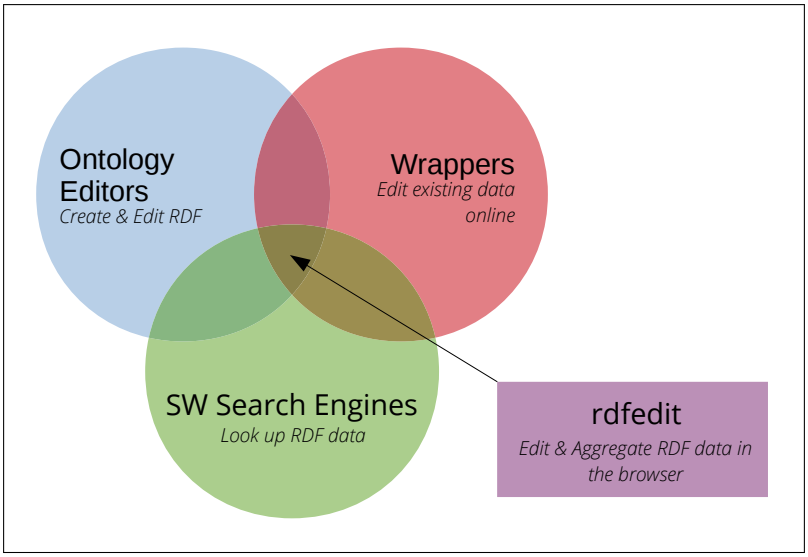

Figure 3.1: The position of rdfedit *in the Semantic Web software space of ontology editors, wrappers and Semantic Web search engines*

rdfedit borrows the basic RDF editing capabilities of ontology editors and combines them with the flexibility of triple store wrapper software into a web application interface, while making use of Semantic Web search engines to simplify and accelerate the creation of triples. It aims at combining a small set of features from each of the three domains introduced – ontology editors, wrappers and search engines – into one application (see Figure 3.1).

This section described the goals and features of *rdfedit*: The web application tries to provide an interface that Semantic Web novices can use to create valid RDF data in a fast and easy manner. The main features of *rdfedit* include bulk-editing, the import and mapping of triples from external into local resources as well as the conversion of literals to URIs. Within the next section, the implementation of *rdfedit* and these features will be discussed in detail.

4 Implementation

While the preceding section is a theoretical description of *rdfedit*'s capabilities, this section focuses on the practical implementation of the hitherto theoretical features. This section starts with introducing the software products *rdfedit* builds and relies upon. It then moves over on how these different software products come together to enable the basic *rdfedit* interface. Furthermore, personal software contributions (own programming code) that implement the main features of *rdfedit* are being discussed.

4.1 Existing Software

rdfedit is built upon three major software components: the web framework *Django*[24], the Python library *RDFlib*[25] and the jQuery plugin *DataTables*[26]. Table 4.1 provides an overview about the purpose of each component, while the subsequent subsections will focus on their role within *rdfedit* with more detail.

Software Component	**Version**	**Purpose**
Django	1.5.1	Web-Server, Basic Architecture
RDFLib	4.1	Handling of RDF Data in the back-end
DataTables	2.0.3	Creation of interactive tables in browsers

Table 4.1: Overview of the main software components rdfedit *builds upon*

4.1.1 Django

Django is a web framework based on the programming language Python, aiming at the quick (and elegant) creation of web applications (cf. Footnote 24). In general, the main intent of web frameworks is the dynamic generation of websites, i.e. computing a view for individual users. Most commonly, web frameworks consist of an interplay between a HTML-template-language that has special placeholder-markups and some computer programs written in a particular programming language that compute values to substitute the placeholders with. When the coding is done, the web-framework-based application hosts itself as web-server and depending on its configuration, the

[24] `https://www.djangoproject.com/`
[25] `https://github.com/RDFLib`
[26] `http://www.datatables.net/`

web application can be accessed only from the developers machine, a specific IP address domain or from any Internet user.

This concept can be easily illustrated by blogs. Blog posts always follow a specific schema (here simplified): there is a title, an author and the text. When users access a specific blog post, the web framework first takes the blog post-template, reads title, author and the text from a database in the background, inserts that information into the template and sends the dynamically generated result document to the users, who can see the fully rendered text in their web browser.

There are other popular web frameworks such as *Ruby on Rails*[27] based on Ruby or *Node.js*[28] based on JavaScript (JS) that follow a similar approach as *Django*. The latter has been chosen in particular because *rdfedit* is intended to be integrated into BBAW's "Digital Knowledge Store" web environment, which is also mainly running on *Django*.

4.1.2 RDFLib

RDFLib is a Python module for working with RDF data. Using that module, RDF graphs can be parsed from all major RDF serializations into abstract graph objects within a Python application. You can then perform operations on that graph object, such as querying the graph object using SPARQL or deleting particular triples.

Since *Django* code is implemented in Python and thus can make use of all additional modules Python can utilize, *RDFLib* seems suitable for the tasks *rdfedit* should accomplish. Pasin [2011] provides an overview of Python modules and Python-based applications that work with RDF data. It appears that many Python modules listed, such as *ORDF*[29] or *Fuxi*[30] are based on *RDFLib* and extend its functionalities. Other modules like *RdfAlchemy*[31] and *Djubby*[32] focus on interacting with triple stores directly - this can also be achieved by *RDFlib* solemnly.

The biggest competitor to *RDFLib* in terms of functions and flexibility seems to be *librdf*[33]. The main difference to *RDFLib* is, that *librdf* delivers programming interfaces to multiple programming languages, among them Python.

When comparing both options, *RDFLib*'s approach on handling RDF data was the preferable choice, since its utilization is easily understandable thanks to an extensive

[27] `http://rubyonrails.org/`
[28] `http://nodejs.org/`
[29] `http://ordf.org/`
[30] `https://code.google.com/p/fuxi/`
[31] `http://www.openvest.com/trac/wiki/RDFAlchemy`
[32] `http://code.google.com/p/djubby/`
[33] `http://librdf.org/`

documentation and provided helping functions needed for the creation of *rdfedit* and its intended features.

4.1.3 DataTables

DataTables is a jQuery plugin for the dynamic handling of tables within a website. Instead of the aforementioned Python-based libraries that run in the back-end and are invisible to the user, JavaScript/jQuery code is used for processing user input and execute appropriate algorithms upon the document on the user's local computer, for example letting some text change its color when a user clicks on it.

It offers multiple advantages over "normal" HTML table elements: While the basic HTML table elements are static and require additional coding to be manipulated, tables based on *DataTables* offer these functionalities out of the box when applied. Alterations regarding content and view of the table, for instance adding new rows or columns or sorting the table for a particular criterion, can easily be applied to a *DataTables* table.

rdfedit will provide a tabular interface for creating and manipulating RDF data. Hence *DataTables* already offers the tools needed for three basic operations: creating new triples/rows, editing triples/rows/cells and deleting triples/rows. When researching for solutions that can simplify the creation and manipulations of tables, *DataTables* was the most prominently suggested, also offering an extensive documentation and code examples. Another promising suitor is *DynaTable*[34] but it lacks functions for the manipulation of its tables' contents.

4.1.4 Basic Interaction Concept

The main building block of *rdfedit* is *Django*. *Django* ties together the web-server architecture, handles Python libraries (e.g. *RDFLib*) that extend *Django*'s capabilities and manages the use of HTML templates and other static resources, like images, JavaScript files (for user interaction with web pages) and CSS files (for website styling). Additionally, *Django* manages databases (e.g. user information), takes care of file uploads and serves files that can be downloaded by a user.

RDFLib is being utilized inside Django. When an RDF graph is being uploaded for further processing by a user, *RDFLib* catches that graph, and extracts all triples from the uploaded RDF graph. The extracted triples are then used to fill a dedicated HTML-table-template with no functionalities, which is then converted into a *DataTable* with

[34] `http://www.dynatable.com/`

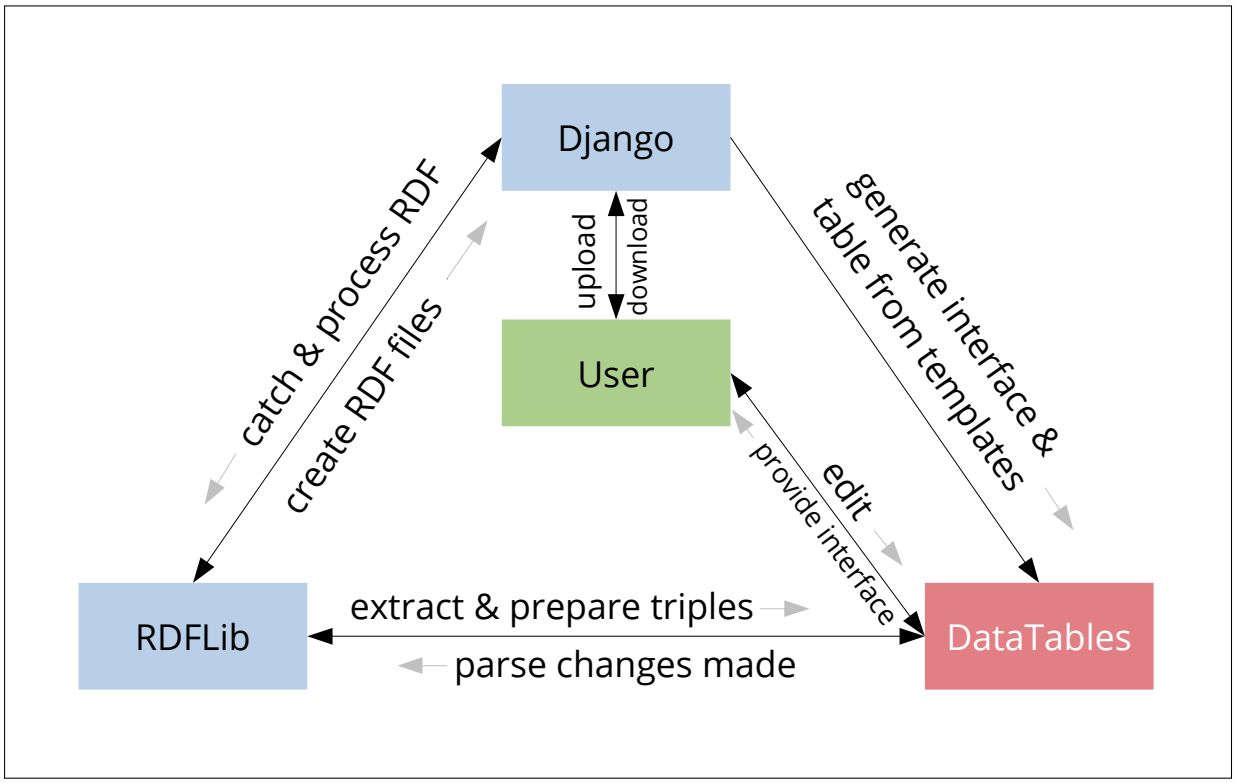

Figure 4.1: Interplay of the three major software components used in rdfedit

more possibilities of interaction. Within that *DataTable* users can apply alterations to the table of triples (henceforth triple-table), thus changing the RDF graph.

When the users have applied all alterations and decide to export the revised graph from the editor interface, the graph within the browser/*DataTables* is sent back to *Django/RDFLib* where it is transformed into a RDF/XML file and served as a download. Figure 4.1 illustrates the concept just described.

You can access *rdfedit* at: `http://141.20.126.167/rdfedit/index/`[35].

4.2 rdfedit Start Page

When accessing the start page of, users are presented with four options on how to begin editing their data. Figure A.1 (p. 95) shows the *rdfedit*'s start page, where the user can choose among the following options:

[35]The source code of *rdfedit* is available at: `https://github.com/suchmaske/rdfedit`. If *rdfedit* seems to be inaccessible, please contact `oliverpohl@ibi.hu-berlin.de`.

(a) *Upload*: Users can choose a local RDF/XML file from their computer and upload it to *rdfedit* for further processing. The RDF file is stored temporarily on the server to extract triples and is deleted afterwards.

(b) *Parse*: Users can enter the URI of a SPARQL endpoint and *rdfedit* reads a limited number of triples from that endpoint, and makes them editable in the browser.

(c) *Example*: Users who only want to try out this web application can choose to let *rdfedit* load an example RDF graph that is being stored permanently and immutably on the server.[36] Changes made in the editing interface to that RDF graph are not saved in the file on the server, so every time other users access the example route, they are being presented with the same graph and triples.

"Triability" is an important feature of new (software) products since it reduces peoples' inhibition in using those products. They don't need to worry whether they will actually be able to use a product. Instead, they can just try it out, notice that they are able to use the product and thus continue to do so in the future [Rogers, 2003, p. 258].

(d) *New*: Users can create a new, empty graph and start adding triples.

The upload, parse and example features function in a similar way. When accessing a RDF file, *Django/RDFLib* executes the code shown in Listing 4.1 to extract all triples from a RDF `Graph` object. Every triple is transformed into an ordered list, where the zeroth[37] Element represents the subject, the first the predicate and the last the object of the triple. That ordered list is then appended to a list of lists, that holds all triples. That list is later utilized to construct the DataTables table.

```
# type(graph) = <class 'rdflib.graph.Graph'>

triple_list = list()

for subject,predicate,object in graph:
  triple_list.append([subject,predicate,object])
```

Listing 4.1: Python code to extract all triples into a list of lists

In case the user chooses to parse triples from a SPARQL endpoint, the number of results is being limited (see Listing 4.2), since most triple stores contain too many triples to be

[36] `http://127.0.0.1:8000/rdfedit/1/spo/`
[37] Lists in Python are zero-based, i.e. the first item of the list has the index `0`.

parsed at once by a normal web server.[38] The data fetched from the SPARQL endpoint is then preprocessed to conform to RDF/JSON in order to be processed by *RDFLib* as a `Graph` object. Then, the same procedure described for uploading RDF files is applied (see Listing 4.1).

```
SELECT *
WHERE {?s ?p ?o}
LIMIT 50
```

Listing 4.2: SPARQL query executed for reading triples from a SPARQL endpoint

Figure A.1 (p. 95) shows the current version of the *rdfedit* start page. When clicking on one of the available buttons, the corresponding actions are performed as stated in the previous paragraphs. When a RDF graph has been processed, these objects among others are forwarded to the *rdfedit* tabular interface (the other objects will be addressed at a later moment):

- **RDF/JSON object**: This object represents the RDF graph using an easy-to-use RDF serialization. Within a browser, it is easier to handle JSON data than XML. Changes made to the tabular interface will be reflected to that RDF/JSON object.
- **Triple-List**: This list represents all triples within a list of ordered lists. It is used to generate the tabular interface of *rdfedit*.
- **Namespace dictionary**: The *rdfedit* settings file contains a namespace dictionary which is used at various points of the application. For once, it helps *RDFLib* to parse RDF graphs by registering namespaces and thus making abbreviated URIs recognizable. Within the user interface it is used to abbreviate long URIs and make them more human-readable (like: `dbpedia:Wil_Wheaton`).
- **Subject/Predicate/Object Sets**: Django builds sets (lists of unique items) for all occurring subjects, predicates and objects. These are passed on to enable auto-completion during the addition of new queries. Since it is very likely that some subjects and predicates (and objects) occur multiple times within a RDF graph, users can start typing the URI or literal, while *rdfedit* filters these sets for substring matches. Furthermore, the short namespace-keys of the namespace dictionary (e.g. `dc:`) are appended to each set. Users can then choose the match of their preference and thus save time when adding new triples.

[38] The limit number is set to 50 for demonstration purposes and can be increased at any time by editing the *rdfedit* source code.

4.3 rdfedit Editing Interface

This section focuses on *rdfedit*'s editing interface and elaborates on all its features and their implementation. Each of the following subsections are devoted to one of *rdfedit*'s major features or components, starting with a general introduction of the interface and then focusing on the details. If necessary, the subsequent paragraphs refer to (pseudo) source code listings to demonstrate how the respective component has been implemented. Among these paragraphs, a suitable choice for a Semantic Web search engine that might contribute *rdfedit*'s set of functionalities will be discussed.

4.3.1 Overview

The most important component of *rdfedit* is its table, which consists of three columns (subject, predicate and object), where each row represents a single triple (cf. Figure A.3, p. 96, marked red). Section 4.3.2 focuses on the creation of that table.

When clicking onto a subject or object cell, it is possible to alter the strings given (cf. section 4.3.5). Such edits can be applied either to a single triple or to all triples containing the altered URI (cf. section 4.3.10). When clicking on a predicate-cell, users are being redirected to the original definition of that predicate (cf. section 4.3.6). There is an icon at the end of each row, that, when being clicked on, induces the deletion of that row/triple (cf. section 4.3.8).

Below the tabular editing area, there are three further table rows. The input fields in the first row allow users to search each column. These are being filtered appropriately while the user types his search keywords into the according column search field.

The second one consists of input masks for the creation of new triples (cf. section 4.3.7 for the detailed triple-addition process, see also Figure A.3, p. 96, marked blue), the input forms and buttons in the last row are used to import triples from external resources, as proposed in section 3.2.3 (cf. Figure A.3, p. 96, marked green). For these functions to work, there is a need of an intermediary service which provides an easy way to search for RDF graphs among the Semantic Web. Section 4.3.11 evaluates two Semantic Web search engines towards their suitability for being used by *rdfedit*, followed by sections 4.3.12 and 4.3.13 which discuss the technical implementation of the triple import and mapping process. Another feature making use of *rdfedit*'s interplay with Semantic Web search engines is a literal-to-URI-conversion, which section 4.3.14 focuses on.

All of the actions possible can be reverted. Undo buttons at the bottom of page and inside the top-bar menue of the interface revert the actions performed last when being clicked on. Section 4.3.9 discusses on how each of the aforementioned possibilities to change the RDF graph can be undone within *rdfedit*.

Each of the following paragraphs will discuss the features and problems described from one to three sides, depending on which components of *rdfedit* are being utilized. First, the user interface itself and the possible interaction are being depicted, followed by how these interactions are being enabled by the JavaScript/DataTables and Django/RDFLib components of *rdfedit*.

4.3.2 Triple-Table Creation

Django. Let's assume a user uploads a valid RDF file on the *rdfedit* start page. Hence, the RDF graph is being processed as described in section 4.2 and Listing 4.1 (p. 27). The graph now exists as a list of lists (triple-list): `[[S1, P1, O1], [S2, P2, O2], ..., [Sn, Pn, On]]` and that list is forwarded to the creation of the editor interface (cf. p. 28).

Django then opens an template file for the generation of a basic and not yet interactive HTML table. Listing 4.3 shows the relevant code snippet from that template file. Django iterates over that triple-list and generates a row with three cells for each list entry. `{% some code %}` indicates Python/Django code that is being executed but does not directly insert a value into the result HTML document. `{{ variable }}` calls a variable and inserts the content of that variable directly into the HTML document which should be rendered inside the user's browser. In this code example, Django creates a new row (`<tr>`) for each `triple` inside the `triple-list` and inserts subject (`triple.0`), predicate (`triple.1`) and object (`triple.2`) into their respective cells.

At this point, *Django* has compiled a HTML document with a static table, only existing on the server-side of the application. That HTML document is then sent to the user's web browser, where some JavaScript/jQuery code is being used to preprocess the static triple-table into an interactive *DataTable*. The latter is then rendered and displayed on the user's screen.

JS/DataTables. While loading the HTML document, jQuery iterates over every entry in the table and uses a dictionary (or hashmap)[39] to abbreviate all URIs for better read-

[39] "Dictionary" is a variable type in Python that contains key-value pairs. The JavaScript (or Java) equivalent to a Python dictionary is called a map, or "hashmap".

```
<table id="triple_table">
...
<tbody>
 {% for triple in triple_list %}
 <tr id="row">
  <td>
   <span id="subject" contentEditable>{{ triple.0 }}</span>
  </td>
  <td>
   <span id="predicate">{{ triple.1 }}</span>
        </td>
  <td>
   <span id="object" contentEditable>{{ triple.2 }}</span>
  </td>
 </tr>
 {% endfor %}
</tbody>
...
</table>
```

Listing 4.3: Django code to generate static rows for the triple tables (simplified for readability)

ability. This dictionary is contained within the *rdfedit* settings file, loaded by *Django* and inserted into the HTML document during its generation (cf. p. 28). For example, all entries in the subject-column that had the long URI `http://dbpedia.org/resource/Wil_Wheaton` would now appear as `dbpedia:Wil_Wheaton`. Listing B.1 and B.2 (cf. p. 103 and p. 103) show the namespace dictionary and the JavaScript code for the abbreviation of the cell contents.

The entries in the object column are analyzed using a regular expression to determine whether the object value is an URI or a literal. Depending on the outcome, a icon[40] representing an URI or a ▪▪▪ icon[41] representing a literal is displayed next to the value. This should facilitate recognizing what kind of value is in an object cell.

Afterwards, jQuery creates a new DataTable by selecting the static triple-table node generated by *Django*. This leads to the table being re-styled and being made interactive. All standard functionalities of a DataTable now apply to the triple-table. Listing 4.4 shows a simplified version of the interactive triple-table initiation. `$("#triple-table")` selects the node with the ID `triple-table`, i.e. the table that has been generated in

[40]The icon downloaded for this document was created by Freepik and published under the Creative Commons BY 3.0 license: `http://www.flaticon.com/free-icon/earth_44027`

[41]The icon downloaded for this document was created by Freepik and published under the Creative Commons BY 3.0 license: `http://www.flaticon.com/free-icon/button-of-three-squares_58896`

```
triple_table = \$('#triple_table').dataTable( {
 "sDom": '<"row"<"span7"l><"span8"i>f>rt<"offset6 span6"p>',
 "sPaginationType": "bootstrap",
 "oLanguage": {
  "sLengthMenu": "_MENU_ records per page",
 }
 "bRetrieve": true
} );
```

Listing 4.4: JavaScript code to initialize the DataTable (simplified for readability)

Listing 4.3. The parameters given here mainly influence the styling of the resulting table.

Finally, the user can access the interactive table with abbreviated cell content and start editing her or his RDF graph. Figure A.2 (p. 96) shows the *rdfedit* tabular interface with the example RDF graph loaded.

4.3.3 Auto-completion

Throughout the interface generation by the server and editing process by the user, *rdfedit* enables the auto-completion of URIs. While the HTML page is being composed inside the users' browsers, the server-side of *rdfedit* computes sets (lists of unique items) for all subjects, predicates and objects contained within the RDF graph that has been uploaded, and sends those sets to the client (cf. also p. 28).

Together with the namespace abbreviations from the namespace-dictionary that is also transmitted from the server, the contents of these sets are used to suggest namespace-prefixes (e.g. `foaf:`) or abbreviated URIs (e.g. `foaf:name`) to users while they are typing to add new triples. It does not matter whether the the value entered by the user matches the beginning, end or middle of a URI.

Let's assume the RDF graph uploaded by a user already contains the subject `dbpedia:Herman_Melville`. Since it is likely another triple with the subject should be created, it suffices to enter "`mel`" into the add-subject-field (cf. Figure A.4, p. 97) and *rdfedit* automatically suggest the abbreviated URI mentioned above.

4.3.4 Filtering Triples

When triples are loaded, it is possible to search for particular triples either by using the table-wide search or applying a column-internal search.

By entering a value in the table-wide search field, DataTables automatically filters and displays those triples that match the entered value (cf. Figure A.5, p. 97). The same applies to the column-specific searches. Below each column is another input field that acts as a column filter. For example, users can easily look for all triples where the predicate is `dc:title` or just use a certain namespace (cf. Figure A.6, p. 97).

The table-wide and column searches are functionalities which belong to the core feature set of DataTables. No further coding needed to be done to make these filtering possibilities work.

4.3.5 Editing triples

Interface. The next paragraphs describe what happens when the user applies changes to a triple. By default, *rdfedit* only allows the alterations of subject and object cells, while predicate cells are immutable. When clicking on a cell in the subject or object column, the cell transforms into an input field, making the content of the cell editable (cf. Figure A.7, p. 98). The cell contents are wrapped in a HTML `<span>` element with a `contentEditable` attribute, which was introduced for the in-browser manipulation of text with HTML5. Having made all changes, the alterations are being saved in the *DataTable* and RDF/JSON object in the background.

JS/DataTables. By clicking somewhere outside the just edited cell, a JavaScript `onblur` event is being called which reflects the changes made onto the RDF/JSON object. For that to happen, *rdfedit* stores the initial value of the edited cell and resolves the abbreviation. Moreover, it also saves the values of the other parts of the triple in order to correctly work with the RDF/JSON object. For a better understanding of the RDF/JSON access pattern within JavaScript, Listing 4.5 shows the structure of such an object for triples with the same subject-predicate-combination only occurring once and multiple times in the RDF graph. Depending on whether a subject or object has been edited, one of the following patterns apply.

(a) **Single Subject Edit**: If there is only one triple with the original subject value, just replace that value. If there are multiple triples with the original subject having different predicate-object-pairs, delete that predicate-object-pair and create a new triple containing the new subject value and that predicate-object tuples (cf. Listing B.3, p. 104 for clarification).

```
{ "ex:Moby_Dick" : {
        "dc:author" :    [{"value" : "ex:Hermann_Melville",
                           "type" : "uri"}],
        "bibo:isbn" :    [{"value" : "978-3800054794",
                           "type" : "literal"},
                          {"value" : "978-1495266997",
                           "type" : "literal"}]
 }
}
```

Listing 4.5: RDF/JSON example (using shortened values for readability)

(b) **Bulk Subject Edit**: First, (a) is applied. In case the change of the subject of one triple should be applied to all other triples having the same original subject-value, copy the predicate-object tuples into the the node of the altered subject and delete all triples containing the original subject-value. To prevent inconsistencies, also change all object-values to the new subject-value, that had the original subject-value before (cf. Listing B.4, p. 106 for alterations). Bulk editing is also being discussed later in section 4.3.10.

(c) **Single Object Edit**: Simply change the object-value of that triple and check, whether the new value is a URI or a literal (cf. Listing B.5).

Before the change is finalized and displayed in the user's browser, the state of the RDF/JSON object is being saved temporarily and information about the change operation ((a), (b) or (c)) are saved in order to be able to revert the changes.

At this point, only changes to the RDF/JSON object, but not the triple-table, have been applied. As you can see in Listing 4.3 (p. 31) and Figure A.3 (p. 96, marked red), the cells in the triple-table consist of more than just plain text. The subject and object cells contain a `contentEditable` attribute and the object cells also contain icons. To maintain the functionality and visual style of these cells, one cannot simply change the plain text content of these cells within the *DataTables* data structure. Instead, HTML code is being generated that also contains the new, altered values and keeps the appropriate attributes and icon information. Listing B.6 (p. 108) contains pseudo-code (simplified code for better readability) to show the creation and insertion of a new object table cell. These HTML containers can then replace the old cells in the triple-table.

Applying some kind of change to a RDF graph in *rdfedit* always induces the following pattern of operations:

(a) Edit RDF/JSON object appropriately.

(b) Save change operation information for potential reverts.

(c) Create and insert HTML container nodes into the triple-table in order maintain functionalities and visual style.

4.3.6 Predicate Vocabulary Look-up

JS/DataTables. While it is possible to alter subject and object cells by simply clicking on them, clicking on a predicate cell invokes something different. *rdfedit* does not allow the change of predicates, since the scope of applied vocabularies for RDF graph predicates is quite narrow when being compared to the vast options of different namespaces that can be applied to subjects and predicates. Ideally, users should only choose among a narrow set of predicates that *rdfedit* administrators find appropriate and match their schema or ontology. Since this might pose an issue regarding the usability of *rdfedit*, section 5.5.6 (p. 75) elaborates further about this problem and suggests possible solutions.

When hovering over a predicate cell with the cursor, a tooltip is generated and displayed which shows the full URI of the predicate. For example, if the predicate is `dc:title`, the tooltip would show `http://purl.org/dc/elements/1.1/title`. If the user clicks on the predicate field, a new browser tab opens and redirects the user to the definition of the predicate, so the user can gain more insight on how this predicate in particular can be used correctly.

4.3.7 Adding Triples

JS/DataTables. When adding new triples to the triple-table only JavaScript and *DataTables* are being involved. The second to last row in the *rdfedit* interface (cf. Figure A.3, p. 96, marked blue) contains three input fields, respectively for the three members of a triple. While users enter new content into the field, *rdfedit* suggests URIs and namespaces that already occur in the graph as they type (cf. section 4.3.3, p. 32). Using these suggestions for auto-completion of URIs should help the users save time and prevent typing errors (see also section 4.2, p. 28). As soon as the user decides to submit the new triple, four things happen:

(a) **RDF/JSON**: The new triple is added to the RDF/JSON object appropriately. Depending on whether a triple with the new subject and/or a new predicate already

exists, *rdfedit* has to perform different operations to insert the new triple into the RDF graph representation (cf. Listing B.7, p. 109).

(b) **triple-table**: Similar to section 4.3.5, HTML nodes containing the new triple and the proper attributes for keeping the functionalities of the cells are being generated and inserted into the *DataTable* triple-table. Listing B.6 (p. 108) shows some pseudo code, that exemplifies the process when a new row is added to the table.

(c) **Operation Stacking**: To revert the addition of the new triple at a later moment, the row-number of that new triple-row is being saved. Hence, when a user decides to undo the addition, *rdfedit* simply deletes the row with that row number. This reversal, i.e. deletion of the new triple, performs as described in the next section (4.3.8).

(d) **Auto-completion Extension**: The new subject, predicate and object are added to their respective auto-completion sets. URIs and literals completely new to the graph will now appear in the auto-completion suggestions (cf. section 4.3.3, p. 32).

To complete the list of basic features of *rdfedit*, the next paragraphs revolve around the deletion of triples within the application.

4.3.8 Deleting Triples

Similar to triple adding and editing, only the JavaScript*/DataTables* part of *rdfedit* is being involved in the process of deleting triples.

JS/DataTables. On the right end of each triple row inside triple-table is a little ⊗ icon[42], that initializes the deletion of that row when being clicked on. For *DataTables*, the deletion process is fairly easy, since it can delete rows by their row-number. When a user decides to delete a triple, three things happen within *rdfedit*:

(a) **RDF/JSON**: The triple is deleted from the RDF/JSON object appropriately. Depending on whether further triples with the same subject or subject-predicate tuple exist within the graph, *rdfedit* performs a different triple deletion operation on the RDF/JSON object (cf. Listing B.8, p. 111).

[42]The icon downloaded for this document was created by Freepix and published under the Creative Commons BY 3.0 license: `http://www.flaticon.com/free-icon/close-delete-remove-button_39`

(b) **triple-table**: *DataTables* deletes the triple/row from the triple-table by the number of that row. Listing B.9 (p. 111) shows simplified code on how a deletion takes place inside the triple-table.

(c) **Operation Stacking**: In case the deletion should be reverted, *rdfedit* saves the triple internally in order to be able to restore it at a later moment. This reversal, i.e. addition of the old triple, is performed as described in the previous section (4.3.7).

4.3.9 Reverting Actions

When performing an action (edit, add or delete) in *rdfedit*, these actions are being saved in the background, invisible to the user. At the bottom of the *rdfedit* user-interface there is an undo button (cf. Fig. A.2, p. 96, bottom right) that initializes the reversal of the last actions performed by the user. Moreover, the persistent top bar of *rdfedit* also contains a tools tab, where the undo action can be found (cf. Figure A.8, p. 98).

JS/DataTables. Depending on the operation the user has executed, more or less information about the affected triple is being stored in an array of dictionaries (here: `action_stack`). Simultaneously, a copy of the RDF/JSON object is stored in another array (here: `rdfjson_stack`). Both stacks always have the same length, i.e. the index numbers of the respective stacks correspond to the same state and action.

In case that operation should be reverted, *rdfedit* calls the appropriate function and information from the most recent object inside the `action_stack` and reinstates the corresponding version of the RDF/JSON object from the `rdfjson_stack`. During the reversal, both entries are deleted from their respective arrays. Table 4.2 lists what information is being stored for which operations and also states the corresponding counter-operations.
So far, the basic manipulation features of *rdfedit* have been addressed. The next pages focus on *rdfedit*'s more sophisticated and unique features, such as bulk-editing, fetching and mapping triples from external resources and a literal-to-URI conversion. These features intend to aid the user in the creation of RDF data in a semi-automatic way by automating time consuming tasks like researching appropriate graphs and extracting triples while leaving the final decision on what will be added the RDF graph with the user.

Operation	Information Stored	Counter-Operation
`Subject Edit`	Old Subject Container, Position, Operation (subject_edit)	`subject_edit(old_subject, position)`
`Object Edit`	Old Object Container, Position, Operation (object_edit)	`edit_object(old_object, position)`
`Bulk Edit`	Old Subject Container, Old Subject Value, New Subject Value, Operation (bulk_edit)	`subject_edit(old_subject)` for all triples with the new Subject Value as the subject; `object_edit(old_subject)` for all triples with the new Subject Value as the object
`Add Triple`	New Row-Number, Operation (add_triple)	`delete_triple(row_number)`
`Delete Triple`	Subject Value, Predicate Value, Object Value, Operation (delete_triple)	`add_triple(subject, predicate, object)`

Table 4.2: Basic operations and counter-operations within rdfedit

4.3.10 Bulk Editing

Bulk editing when working with RDF means the application of changes to multiple triples at once. For this operation to happen, only JavaScript code has to be evoked.

JS/DataTables. When altering a subject inside the *rdfedit* interface, an ↞ icon[43] appears inside the subject cell (see Figure A.9, p. 98). Clicking on that icon causes *rdfedit* to iterate through the subject column and alter all subject cells, that have the initial value of the subject with the "apply to all icon" next to it.

The same applies to the object column: Since it can happen that a subject URI is being referenced as an object in other triples, also applying changes to objects of the same value as the original subject keeps the RDF graph consistent. However, it is not possible to initiate a bulk-edit by altering an object because object values are less likely to appear as subjects within the same graph multiple times.

This operation affects *rdfedit* on multiple instances:

(a) **RDF/JSON**: As described on in section 4.3.5 (p. 33), all predicate-object tuples belonging to the original subject value are extracted and inserted into the JSON

[43]The icon downloaded for this document was created by Dave Gandy and published under the Creative Commons BY 3.0 license: `http://www.flaticon.com/free-icon/reply-arrow_25691`

hierarchy of the new subject value. The original subject node is then being deleted. Afterwards, all objects with the initial subject value are changed too.

(b) **triple-table**: During the initial single-subject change, *rdfedit* generated a HTML container with the new subject value for the triple-table and writes it into the cell. When using bulk edits, that same HTML container is injected into every cell having the original subject value. *rdfedit* takes the newest item from the `action_stack` (see p. 37) to access the value of the original value for comparing it with the other cells. Moreover, *rdfedit* generates corresponding HTML containers for object cells and applies the replacements in the object column as well.

(c) **Operation Stacking**: Although having to make two clicks to execute a bulk edit, doing so should be rather viewed as a singular action than two separate ones. Clicking on the bulk-edit icon makes *rdfedit* use (and remove) the newest items from the `action_stack` and `rdfjson_stack`. When having applied the change, a new object containing information on how to revert the bulk edit is being stored instead of the single-subject edit. In case of a reversal, another bulk edit is applied, where the old value substitutes the newer values inside the RDF/JSON object and triple-table cells.

4.3.11 Triple Import via Semantic Web Search Engines

Whereas all other operations were only executed inside the browser, i.e. just involving the client side, importing and mapping triples with *rdfedit* also utilizes *Django*, i.e. the server side.

The idea behind these functions is to enable the reuse of already existing data when creating new instance metadata in RDF by importing such data from external triple stores, extracting relevant data and mapping them into your local RDF graph.

Requirements. In order to fetch Semantic Web data from external resources in an efficient way, one may not rely solely on SPARQL. For once, sending queries to multiple triple stores at once requires a deeper knowledge about these databases, their use of heterogeneous data, domains, vocabularies and ontologies while having to be able to take all these factors into consideration when constructing a federated SPARQL query. Hence it is easier to use a dedicated Semantic Web service that has indexed data available from publicly accessible triple stores and search the index of that service instead.

The following requirements for such a service have to be met in order to be used by *rdfedit* and thus the user efficiently:

(a) **Index Size**: The service should have incorporated data from publicly available triple stores across many domains.

(b) **Searchability & API**: The service should provide means to perform searches on its index without having to use SPARQL. Instead this service should provide an API that makes constructing particular queries easier to write and to execute, e.g. using HTTP requests (REST API). Moreover, this API should offer possibilities to refine the results by domains (e.g. dbpedia), occurring classes (e.g. `foaf:person`) and data format (e.g. RDF or RDF/XML).

(c) **Return Data**: When retrieving a result list, this list should be given in either JSON or XML instead of a HTML representation. The first two formats make it easier to process the result list data. Additionally, the result list should contain the URIs of the graphs that contain matches to the query.

(d) **Currentness**: The service should ingest new and update existing data regularly, so *rdfedit* is able to import correct and topical relevant triples.

Section 3.3.3 (cf. p. 20) gave a brief introduction about the two Semantic Web search engines *Sindice* and *Watson*. Both have indexed a large amount of triples across various domains and offer some level of API support. The next paragraphs analyze Sindice and Watson towards their suitability to meet the aforementioned requirements.

Sindice. Sindice[44] was initiated in 2008 as a result of inter-European collaboration of research institutes such as DERI[45], Fonanzione Bruno Kessler[46] and OpenLink Software[47] with the goal to provide a lookup service for Semantic Web data and thus facilitating their accessibility.[48]

(a) **Index Size**: According to Sindice's own statistics[49], the search engine has incorporated triples with more than 1350 million unique URIs across around 1000 different namespaces.

[44] `http://sindice.com/`
[45] `https://www.deri.ie/`
[46] `http://www.fbk.eu/`
[47] `http://www.openlinksw.com/`
[48] `http://www.w3.org/2001/sw/wiki/index.php?title=Sindice&oldid=1125`
[49] `http://sindice.com/stats/basic-stats/#`

(b) **Searchability & API**: Throughout active development, three versions of APIs have emerged, `v3` being the most recent one.[50] Using this REST API, developers can execute keyword or triple queries and filter results by several facets such as utilized ontology, occurring classes, occurring predicates and available formats (e.g. RDF/XML). Sindice can then be queried by constructing an URL with all query parameters and then make a HTTP request.

(c) **Return Data**: Sindice can return result data in three different formats: JSON, Atom XML and RDF/XML.

(d) **Currentness**: Unfortunately, the running Sindice has announced to terminate the support of their platform by end of 2014 [Franzon, 2014]. Nevertheless, the service will still be hosted but it remains unclear, whether new data will be added to Sindice's index.

(e) **API example**: Send a query to Sindice to obtain information about *Herman Melville* who has to be of the class `person`. The result data should be returned in a JSON document.[51]

It is questionable whether Sindice is suitable for looking up Semantic Web data in the future because of the project's end of support. However, by offering a useful and developer-friendly API for querying and obtaining results from a large index, Sindice fulfills three out of the four listed requirements.

Watson. The other Semantic Web search engine Watson[52] was launched in the same time frame as Sindice. Starting in 2007, Watson began to amass RDF graphs and make them searchable. Similar to Sindice, Watson should rather be viewed as a "gateway to the Semantic Web" [d'Aquin et al., 2011] than a Semantic Web search engine.

(a) **Index Size**: There doesn't seem to be current statistics about Watson's index size. According to its statistics page[53], the service has stopped compiling statistics about its own data. The most recent data about Watson's index size originate from 2007 from d'Aquin et al. They report Watson having indexed about 25500 unique documents and 1.1 million distinct URIs. Running a few test queries on Watson reveals that this service lacks data from the Semantic Web focal point, the DBpedia.

[50] `http://sindice.com/developers/searchapiv3`
[51] `http://api.sindice.com/v3/search?q=Herman+Meliville&fq=class:person&format=json`
[52] `http://watson.kmi.open.ac.uk/WatsonWUI/`
[53] `http://watson.kmi.open.ac.uk/Demo%20and%20Stats.html`

(b) **Searchability & API**: Watson offers query refining possibilities to a minor extent. You can choose whether your query keywords should match classes, properties or individuals within RDF graphs. Unfortunately, it is not possible to set the domain the RDF graphs should stem from.

These functionalities are also reflected inside Watson's APIs. Watson offers a direct interface for Java programmers and also provides a REST API.

(c) **Return Data**: Querying Watson by utilizing the REST API will return a XML document containing the results. The document cannot be retrieved as a JSON representation.

(d) **Currentness**: The development of Watson seems to be halted, if not stopped entirely. Since 2011 there haven't been any major publications mentioning the active usage or further development of Watson. Some components of the Watson website also do not function. For example, the RSS news feed cannot be loaded and some image components of the HTML are missing.[54] Additionally, the statement about having stopped collecting statistical data about the search engine indicates the termination of the active service (cf. footnote 53).

Examining Watson more closely raises many issues for its interplay with *rdfedit*. Not only is Watson's index small, it also lacks major Semantic Web data. While the search itself seems functional, Watson does not seem to offer the granularity needed for *rdfedit*. The biggest concern is that the support for Watson seems to have been terminated a few years ago.

Semantic Web Search Engine Comparison. Now that both search engines have been examined, a direct comparison of Sindice and Watson will estimate, which one is suited better to work within *rdfedit*. Table 4.3 lists the requirements mentioned in the beginning of this section (cf. p. 40) and assigns an indicator of how good the respective service seems to fulfill each point[55]:

The comparison in Table 4.3 shows that Sindice is more eligible for being used by *rdfedit* than Watson. Sindice's index and features surpass Watson's capabilities of being a Semantic Web search engine and offering interfaces for developers. Still, it is worrisome that both services will likely not be maintained in the future. No matter which search

[54] `http://watson.kmi.open.ac.uk/Overview.html`

[55] + for meeting the requirements completely; o for meeting some of the requirements; - for meeting the requirements insufficiently or not at all

Requirement	**Sindice**		**Watson**	
Index Size	Large Index, Multiple Domains	+	Small Index, Missing important domains like DBpedia	–
Searchability & API	Multiple refinement possibilities, extensive REST API	+	Some refinement possibilities, no domain constraints, REST API	o
Return Data	Multiple return formats: XML, JSON, RDF/JSON	+	Only returns XML	o
Currentness	End of support in 2014, still having current data	o	Development and support already stopped (around 2011-2012)	–
Overall	Meets most requirements	+/o		

Table 4.3: Comparison of Sindice and Watson towards their suitability for rdfedit

engine will be used in *rdfedit*, a longer lasting solution has to be estimated or developed. Section 5.5.9 discusses the implications of using a no-longer-maintained service and gives an outlook on how that issue can be tackled.
However, Sindice is a suitor for *rdfedit* regarding short-term implementation and demonstration purposes. Since the market of Semantic Web search engines is quite thin, *rdfedit*'s triple-fetching, triple-mapping and literal-to-URI-conversion will rely on Sindice for the time being.

4.3.12 Triple Fetching

The basic editing functions described earlier only utilized JavaScript code, i.e. only processed data on the users' computers. Features connected to Sindice also use server-side computing via Django. This procedure includes two major steps: a) Getting URIs of RDF graphs that match *rdfedit*'s users needs, discussed in this section, and b) extracting and mapping triples from those graphs into the one the users work on inside their *rdfedit* browser interface, described in the next section (4.3.13).

Making use of Sindice's search engine capabilities, *rdfedit* should execute semi-preconfigured queries to obtain graphs containing relevant data for the user. Administrators of *rdfedit* can create query-presets on certain types of data that should be re-

```
{
"Person" : {
        "fq=domain" : "dbpedia",
        "fq=class" : "foaf:person"},

"Location" : {
        "fq=domain" : "geonames",
        "fq=format" : "RDF"}
}
```

Listing 4.6: Triple Fetching configuration for the `Person` and `Location` preset

trieved. These query-presets include parameters offered by Sindice's API. All the user has to do is to choose among the existing presets (e.g. Person, Location, Piece of Art) and enter query keywords that describe their needs.
Listing 4.6 shows query-presets for the type `Person` and `Location`. The first configuration, labelled `Person`, tells *rdfedit* to construct a query-URL, so that Sindice should only look up RDF graphs originating from the DBpedia that contain the class `foaf:person`. When looking up a `Location`, data from geonames.org should be retrieved. Since geonames is likely to return tabular HTML data, the configuration specifies to obtain RDF data, so triples can be extracted directly from the graphs.

The process of fetching triples via Sindice requires the interplay of Django and JavaScript at multiple instances and can be roughly summarized as followed (also cf. Figure A.10, p. 99):

Django. When generating the HTML page, the triple fetcher configuration file is being accessed to read the labels of the presets (here: `Person` and `Location`) and insert those strings in the dropdown-list-button where users can choose a query-preset (see Figure A.11, p. 100, marked blue).

JS/DataTables. The last row on the *rdfedit* user interface (cf. Figure A.3, p. 96, marked green) is used to invoke the triple fetching mechanism (cf. Figure A.11, p. 100, marked red). On the left side, users can enter query keywords, in the middle they can choose among given presets and the buttons on the right side are there to execute the query mechanism and to choose among the list of result URIs.

Pressing the `Fetch Graphs`-button invokes the browser to submit the entered query keywords and chosen query preset to Django using an AJAX callback function. Asynchronous JavaScript and XML (AJAX) is designed to allow the dynamic and asynchronous exchange of small chunks of data between clients, e.g. a web browser on a personal computer and web applications, i.e. servers, and add content dynamically to a web page without having to reload that website. The data sent to the server is processed in some way and induces the server to sent some kind of result data to the client.

For example, users click on a thumbnail image of a product in an online-shop. By doing so, the browser-client requests more information about that product (e.g. manufacturer, available colors and size dimensions) from the server, which looks up that information in an internal database and returns that data in a form, the client can work with. The browser receives that data and injects it into the web page, in order to be viewed by the user.

Django. Django receives those parameters (keywords and preset) and constructs a query URL implementing the following procedure[56]:

(a) Take the base URL of the Sindice API[57].

(b) Append `q=<keywords>&` to the API-URL.

(c) Look up the configuration for the chosen preset.

(d) For each key (parameter) in that preset, append `<parameter>=valueOf(<parameter>)&` to the API-URL.

(e) Append `format=JSON` to the API-URL to ensure the return data from Sindice is a JSON document.

(f) The following query example is an URL to request relevant RDF graphs about `Herman Melville` coming from the DBpedia having the class `foaf:person`. The data should be returned as JSON:
`http://api.sindice.com/v3/search?q=Herman+Melville&fq=domain:dbpedia&fq=class:foaf:person&format=json`

[56] The full source code of that procedure can be found in Listing B.10 (p. 112)
[57] `http://api.sindice.com/v3/search?`

When the query construction is done, a query to Sindice is sent by *Django*. Sindice returns a JSON document containing metadata about graphs that match the query (cf. Listing B.11, p. 113 for an example), including the URIs to access those RDF graphs. The matches listed in the result document are sorted by their respective ranks assigned by Sindice.

Note that Sindice does not return the RDF graphs themselves, for their state in Sindice's index might not always be congruent to their actual representation in their respective original data bases.

Django extracts the URIs of the matching graphs and transfers them to the user using AJAX again.

JS/DataTables. The result URIs are injected into the `Choose Graph` button as a drop-down list. When users click on that button after they have invoked a query, they can choose an URI among the results by clicking on that button (cf. Figure A.12, p. 100). If they have made their choice and clicked on the URI that seems to fit their needs best, *rdfedit* starts to extract triples from the graph behind that URI and map them into the local graph appropriately.

4.3.13 Triple Mapping

JS/DataTables. At this point, the user has initiated the triple fetching process described in the previous section and has chosen one of the URIs that were retrieved via Sindice. The client-side of *rdfedit* then sends the selected URI and query-preset-label back to the server-side using an AJAX callback function.

Django. On the server-side, Django receives the URI and query-preset-label and fetches the corresponding RDF graph. Django then utilizes RDFLib to read that graph and store it in a variable for easier processing within the Python runtime environment.

Next to the query-configuration file mentioned earlier (cf. Listing 4.6, p. 44) there is another JSON-formatted configuration file that determines what triples should be extracted from the remote RDF graphs and how those extracted triples should be mapped into the local graph. Listing 4.7 shows the mapping-configuration for the same examples as before: `Person` and `Location`.

For the mapping process being able to work properly, both the query-configuration file and the mapping-configuration file have to contain the same top-level key-items;

```
{
 "Person" : {
  "dbpprop:author" : "dc:creator",
        "foaf:name" : "foaf:name"},

 "Location" : {
        "rdfs:isDefinedBy" : "dcterms:spatial"}
}
```

Listing 4.7: Triple Mapping configuration for the `Person` and `Location` preset

in this case `"Person"` and `"Location"`. Depending on the query-preset-label chosen earlier, *rdfedit* can determine what triples should be extracted from what kind of graph.

The configuration file tells *rdfedit*'s server-side to extract all triples with predicates are equal to a key contained within the subdocument for the query-label. The predicates of those triples are then transformed into the corresponding values to those JSON keys. To extract those triples, Django generates SPARQL queries as shown in Listing 4.8 and submits them the RDFlib graph object.

```
new_graph = Graph() # RDFlib graph
external_graph = Graph() #RDFlib graph
external_graph.load(URIRef(graph_uri))

for orig_uri in type_mapping:
  sparql_query = 'select␣*␣where␣{␣?s␣' + orig_uri + '?o␣.}'
  for row in external_graph:
          new_graph.add([row.s, new_uri, row.o])
```

Listing 4.8: Extracting and re-mapping triples from external RDF graphs (simplified Python code)

From our previous query-example we know the graphs being fetched for `Person` are lying exclusively within the DBpedia triple store. Hence, one can make use of the structure particular graphs underlie at certain domains. In this example, all triples containing the predicate `dbpprop:author` or `foaf:name` are read from that graph. Predicates being `dbpprop:author` are changed to the more general term `dc:creator`, while triples with `foaf:name` as a predicate remain unchanged. Table 4.4 compares the original triples from the DBpedia graph about Herman Melville and their mapped derivations towards the `Person` preset.

Along this whole process, a query to Sindice requesting relevant RDF graphs for a particular query-preset and keywords has been issued and one graph among the results has

[58] `http://dbpedia.org/page/Herman_Melville`

Original Triple	Mapped Triple
`dbpedia:Herman_Melville dbpprop:author dbpedia:Moby_Dick .`	`dbpedia:Herman_Melville dc:creator dbpedia:Moby_Dick .`
`dbpedia:Herman_Melville foaf:name ‘‘Herman Melville’’ .`	`dbpedia:Herman_Melville foaf:name ‘‘Herman Melville’’ .`

Table 4.4: Original and mapped triples for Herman Melville's entry in the DBpedia[58] (excerpt). Namespace declarations were omitted for readability.

been selected by the user. Triples with predicates listed in a mapping-configuration file have been extracted and remapped according to that configuration. During the extraction process, a new RDF graph is being formed. Since JavaScript can't directly work with RDFlib graph objects, all triples are transformed into a list object, similar to the process shown in Listing 4.1 (cf. 27). Lastly, that list is sent back to the client side and further processed by JavaScript and DataTables within the users' browsers.

JS/DataTables. Receiving the triples fetched and mapped from Sindice invokes the triple addition function inside the JavaScript/DataTables framework described in section 4.3.7, following many programmers' paradigm of not repeating source code. *rdfedit* creates new HTML containers for each part of the triple and inserts those into the triple-table. Because the basic triple adding function is reused, the import (i.e. addition) of those new triples are also reversible.

As Table 4.4 shows, *rdfedit* only maps the triples' predicates. It is left to the users to manually alter the subject and object URIs or leave them unchanged entirely. Subject URIs might need to be changed in order to conform to a specific pattern or naming convention, like `ex:Lastname_Firstname_ID` instead of DBpedia's `dbpedia:Firstname_Lastname` scheme. Object URIs on the other hand might not need to be changed at all, for some triples should refer to external triple stores on purpose, thus practicing Linked Data reuse and utilizing the capabilities of the Semantic Web. The respective operations have been described in section 4.3.5 (cf. p. 33) and 4.3.10 (cf. p. 38).

In the end this graph-fetching operation and subsequent triple-mapping have influenced *rdfedit* at different instances:

(a) **RDF/JSON**: New triples have been obtained from an external resources and added using *rdfedit*'s own triple addition function. Simultaneously, those new triples are added to the internal RDF/JSON object.

(b) **triple-table**: New HTML containers have been generated and inserted into the triple-table. Users now can perform further actions on these new triples inside the *rdfedit* interface.

(c) **Operation Stacking**: When the client receives the list of fully mapped triples from *rdfedit*'s server side, the client loops over that list and adds each new triple to the table one by one. Hence the addition of every new triple is reversible on its own. However, when a reversal is applied, it is not fully clear which triple was added most recently and thus having the potential to confuse or frustrate users. Section 5.5.3 (p. 73) elaborates more deeply of the benefits and disadvantages of undoing multiple triple additions in a one-by-one manner.

4.3.14 Literal-to-URI-conversion

Another process closely related to the triple fetching and mapping that have been described previously is the literal-to-URI conversion feature of *rdfedit*. The purpose of this feature is to automatically find and suggest URIs for triples added by the user, where the object is a literal. The user can choose among a list of possibly matching URIs and the URI selected by the user then replaces the literal in the same triple within the user's RDF graph.

This functionality should save the user some time when researching for URI that might fit their needs by semi-automating that research process and only needing to make a single selection.

Like the earlier elaborated features, the literal-to-URI-conversion both requires the client- and server-side of *rdfedit* to interact with each other. Figure A.13 (p. 101) summarizes this process as a flowchart.

JS/DataTables. When the user adds a triple inside the *rdfedit* interface, the web application checks whether the object of that triple is a URI or a literal. In case the object part of that triple is latter, both the predicate and object value are sent back to the server via AJAX.

Django. Section 4.3.12 (cf. p. 43) introduced the query-presets and the Sindice query-configuration file, as the previous section did with the mapping-configuration files. As soon as the server-side of *rdfedit* obtains the values submitted by the user, it iterates through the JSON mapping-configuration file on the second level, i.e. ignoring the

preset-labels at first. When the submitted predicated matches one of the JSON keys or values inside the mapping configuration, Django takes the preset-label of that node and uses the query-configuration of the same label to construct and issue a query for Sindice. In case the predicate appears in more than one configuration, multiple queries are issued.

Sindice delivers JSON documents containing the URIs of possibly relevant graphs and respective metadata (cf. Listing B.11, p. 113 for a detailed example). All the *rdfedit* server does now is to extract only the result URIs and transmit them to the client, where the users can select the URI they think is most appropriate.

JS/DataTables. As soon the client receives that list of URIs, *rdfedit* creates a dropdown menu and inserts it directly next into the object cell of the row that has just been added by the user (cf. Figure A.14, p. A.14). Selecting an URI from that dropdown menu takes influence on *rdfedit* at three different levels:

(a) **RDF/JSON**: By choosing one URI to replace a literal, *rdfedit* calls its object edit function to substitute the literal object value with the selected URI. This leads to *rdfedit* also changing the object value inside the RDF/JSON object.

(b) **triple-table**: Simultaneously *rdfedit* creates a new HTML container for the substitute-URI-object and inserts it into the according row inside the triple-table. This overwrites the complete contents that existed previously in that object cell and thereby also deletes the dropdown menu.

(c) **Operation_Stacking**: When the conversion takes place, users have added a new triple beforehand. That triple is added normally using *rdfedit*'s respective functions. When the conversion happens, *rdfedit* performs an edit, hence making the change from literal to URI fully reversible. Still, reversing the conversion cannot bring back the dropdown menu holding matching URIs for that literal. This issue is also being discussed in section 5.5.3 (p. 73) later on.

At last, all features able to influence the *rdfedit* interface and RDF graphs have been thoroughly described. After having applied all desired alterations, users can export their triple-table to an RDF/XML file.

4.3.15 Exporting RDF Files

In order to export and download the edited graph, you can click on the export-button (cf. Figure A.2, 96, bottom left) or use the option in the top bar (cf. Figure A.8, p. 98). During the editing process, all changes have been applied to the triple-table and the RDF/JSON object. Both instances have been managed only on the users' local machines.

The client side of *rdfedit* opens a connection to the server and transfers the RDF/JSON object to Django. On the server, the client serialization (RDF/JSON) is being transformed into RDF/XML. RDFlib, the Python library specialized on handling RDF data, receives the client serialization of the graph, forms it into a RDFlib graph object and finally re-serializes it the RDF/XML representation of the graph. Django then creates a new XML file and fills it with the contents of the RDF/XML object. Afterwards, that file is transferred back to the client where the users can choose to download the RDF/XML onto their hard-drives.

This action affects *rdfedit* on the following instances:

(a) **RDF/JSON**: As soon the export is initialized, the RDF/JSON object is uploaded to the Django-server. Meanwhile, its content and representation on the client-side remain unchanged.

(b) **Django/RDFLib**: Django temporarily stores the received RDF/JSON object, transforms it into RDF/XML and serves it to the user. For the sake of downloading, a copy of the RDF/XML document has to be saved inside *rdfedit*'s internal server-database. Once the download process has been finished or cancelled, that database entry is erased. This circumstance could also be used to save RDF graphs on the server. For demonstration and performance purposes, this has not been implemented yet. This circumstance is further addressed in section 5.5.4 (p. 74).

Throughout this whole section the user-interface and functions of *rdfedit* have been presented. While the first half of this section discussed the basic features such as adding, editing and deleting triples, the second half focused on user-aiding functions like graph fetching, triple mapping and literal-to-URI-conversion that represent a semi-automatic aid for users for the creation of RDF data.

5 Evaluation

The last section presented *rdfedit*'s setup, workflows and features inside the final product. So far problems during the implementation of the web application, conceptual concerns and further not (yet) implemented features for *rdfedit* have been mainly omitted.

This section evaluates the implementation and usability of *rdfedit* by performing a heuristic usability test in section 5.2 and describing the setup and results of a Thinking Aloud usability test in section 5.3, followed by further self-criticism in section 5.5 where issues that arose during the implementation of the web application and features that have not yet made it into the current version of *rdfedit* are discussed.

5.1 Usability Testing

rdfedit was implemented to help people who know only very little about Semantic Web technologies create RDF data but it hasn't been established yet whether the web application succeeds or fails chasing that goal. To find out whether it can be easily used as intended during development, tests to examine the *usability* of *rdfedit* were conducted. Simply put, "usability is the question of how well users can use [a] functionality" [Nielsen, 1994, p. 25] of a given program or system.

Evaluating the usability of a program is important, since the way developers think their software should or will be used not always concurs with the way users actually operate it. Usability engineering is a process to examine how a program is being used and especially determine the problems that hinder users working with a piece of software productively. The knowledge gained throughout that process is then used to improve the software and make it more usable for the users. Testing the usability and implementing improvements is a circular process that has to be repeated often to create a software that fulfills its purpose and satisfies users' needs.

Improving the usability of software most often leads to higher productivity and lower costs in a productive environment, since a usable piece of software is easier to learn and faster to operate, thus saving time, hence managing to accomplish more tasks in the same time compared to an unusable software [Nielsen, 1994, p. 2-7]. Even more important than efficiency of a software is that its users are content with using it. A well usable software ensures higher acceptance within its potential user base and thereby influences the adoption of the software positively [Möller, 2010a].

Usability tests were conducted for this thesis. It is not quite eligible to call the conducted tests a whole "Usability Engineering Process" because they only focus on examining peoples' behavior when they are using *rdfedit*. This thus refers to "Usability Testing".

5.1.1 Choice of Usability Test Methods

There is a large variety of methods to conduct usability tests. Nielsen [1994, p. 224] created an overview listing all common usability testing methods including their advantages and disadvantages (cf. Table 5.1).

Method Name	**Users Needed**	**Main Advantage**	**Main Disadvantage**
Heuristic Evaluation	None	Finds individual usability problems. Can address expert users.	Does not involve real users, so does not find "surprises" relating to their needs.
Performance Measures	< 10	Hard numbers. Results easy to compare.	Does not find individual usability problems.
Thinking Aloud	3-5	Pinpoints user misconceptions. Cheap test.	Unnatural for users. Hard for expert users to verbalize.
Observation	< 3	Ecological validity; reveals users' real tasks. Suggests functions and features.	Appointments hard to set up. No experimenter control.
Questionnaires	< 30	Finds subjective user preferences. Easy to repeat.	Pilot work needed (to prevent misunderstandings).
Interviews	5	Flexible, in-depth attitude and experience probing.	Time consuming. Hard to analyze and compare.
Focus groups	6-9 per group	Spontaneous reactions and group dynamics.	Hard to analyze. Low validity.
Logging actual use	< 20	Finds highly used (or unused) features. Can run continuously.	Analysis programs needed for huge mass of data. Violation of users' privacy.
User Feedback	Hundreds	Tracks changes in user requirements and views.	Special organization needed to handle replies

Table 5.1: Summary table of usability testing methods taken from Nielsen [1994, p. 224]

Because there is no user base of *rdfedit* yet, several of the methods listed in 5.1 are not suitable to evaluate the usability of *rdfedit*. For instance, analyzing massive user feedback, evaluating questionnaires or performing log file analyses would require a certain

number of people already using *rdfedit*. The same circumstance is applicable to performance measures. This method would presume that people already have worked with *rdfedit* productively, either to measure their improvements on producing RDF data or giving insightful information about their problems they have with the software. Similarly, observing peoples' behavior when they are using *rdfedit* can also not be deemed as an appropriate method, since this method would also require *rdfedit* being in a productive environment while monitoring the user. In-depth interview only require a small number of people of an established user-base, which does not exist for *rdfedit* for the time being.

Apart from the user base, time and finances are further limiting factors that reduce the number of possible choices. To gather enough people to participate in a usability test it might have been necessary to offer incentives. Forming focus groups is as an unattainable goal for this thesis, since at least twelve people would have needed to be recruited in order to create two groups. Setting appointments with each group repeatedly and acquiring a room with all equipment necessary would have been not realistic.

In the end, two usability test methods were chosen to evaluate the usability of *rdfedit*: Heuristic evaluation and thinking aloud tests.

Performing a heuristic evaluation does not require many financial or personal resources and is yet quite efficient. Nielsen [1994, p. 156] estimates that 35 per cent of all potential usability problems can be found using a heuristic evaluation by only a single evaluator. This evaluation method consists of a small set of rules (heuristics) that should be used to identify usability-problems users might encounter [Möller, 2010b]. Hence, developers of an application can check themselves whether they followed the rules during their development [Nielsen, 1994, p. 155].

Molich and Nielsen [1990] provided an early example of software usability heuristics which still can be applied today: provide simple and natural dialogue (i.e. interaction), apply consistent designs and interaction models, provide feedback to the users' inputs etc. Making use of this method brings the advantage that the heuristics represent the consensus and standard aspects of a computer application that are expected by the everyday user. More importantly, heuristics build upon knowledge other usability experts have already compiled and thoroughly tested.

Since the introduction of heuristic evaluation, many heuristics have emerged that tackle different aspects of computer applications. For example, Budd [2007] proposed "Heuristics for Modern Web Application Development" which will be used to evaluate the usability of *rdfedit* heuristically.

The second usability testing method conducted for the evaluation of *rdfedit* is a thinking aloud test. During a thinking aloud test, participants verbalize their thoughts and actions while they are using a piece of software to solve a given task [Lewis, 1982]. Through this method, you can gain valuable information about the participants' thoughts, expectations and feelings when using a software. However, you do not measure an objective but rather individual, subjective usability for each participant [Rogers et al., 2011].

For example, if participants were asked to add a triple using the *rdfedit* tabular interface, they should narrate their every action on the screen and their intentions behind it. "I am moving the mouse cursor to the 'Add Subject' field in order to enter a new URI" would be a fitting exemplary statement by a participant.

Since participating in a Thinking Aloud test puts people in the unnatural situation of narrating their thoughts, intentions and actions, some participants might feel inhibited or stressed. That circumstance could lead to mistakes or inaccuracies which might defer the result of this test [Holzinger, 2005]. Moreover, participants often construct their own theories why some of the features tested won't work or could be improved and thus deliver input to the software developer that is not always useful [Nielsen, 1994, p. 195].

The main advantages of a Thinking Aloud test is that this method can be applied with a small number of people and yet return insightful results. Since the participants express what they do and why they perform certain actions, software developers gain insight about whether the software can be used intuitively and whether there are misconceptions that lead to problems while using that software [Holzinger, 2005].

5.2 Heuristic Evaluation of rdfedit

The heuristics used to evaluate *rdfedit* originate from Budd [2007] and focus especially on web applications. His "Heuristics for Modern Web Application Development" are founded on the canonical usability principles of Molich and Nielsen [1990]. The next paragraphs will compare Budd's to the current state of *rdfedit* and estimate problems people might encounter while using *rdfedit*.

1. Design for User Expectations. When developing a web application, programmers should use common web conventions that users are accustomed to and not surprise them negatively. Ideally, the web application should feel like if people were using an

"offline application". *rdfedit* uses the jQuery and DataTables framework for displaying the triple-table. These software libraries enable the creation of an interactive table inside the browser. The team behind DataTables constantly tests their software so websites utilizing DataTables will be "free of surprises".[59] Hence, *rdfedit* complies with this heuristic.

2. Clarity. Interactive parts of a web application interface (e.g. buttons and input fields) should be labeled and have meaningful icons that resemble their actions. *rdfedit* follows that heuristic by labeling all buttons according to their action and uses an appropriate icon the emphasize the operation that button performs. For example, the `undo` button is labeled as such and also contains a ↺ icon.[60] Additionally, all input fields are labeled (cf. Figure A.2, p. 96). Considering the clear labels on what element in the interface does what task, *rdfedit* complies with this heuristic.

3. Minimize Unnecessary Complexity and Cognitive Load. This heuristic demands to remove unnecessary features from the application, break down complicated processes into multiple tasks and to prioritize functionalities by using alignment, shape and color. Buttons that invoke an instant operation such as adding triples or reversion actions are held in a dark blue. The `fetch graphs` button is held in a lighter blue since it is only part of the triple fetching process. The dropdown-menu-button which users can select triple-fetching-presets and the accordingly fetched graph URI from have a grey background color. Searching within each column, adding triples and importing data from external resources all have their own column in the tabular interface and thus separate themselves but also align themselves with the table content. Regarding these design decisions, *rdfedit* also complies with this heuristic.

4. Efficiency and Task Completion. Budd suggests to streamline an user interface towards the tasks that are most commonly performed. *rdfedit* only contains a narrow set of RDF graph manipulating features, so all these functions can be used without needing to activate them somewhere else externally. Additionally, there should be separate options to activate an advanced mode for expert users and options to customize the user interface towards their own preferences. Since *rdfedit* is aimed towards Semantic Web novices, there are no advanced options yet. To add such functionality or customize the

[59] `http://www.datatables.net/development/testing`

[60] The icon downloaded for this document was created by Dave Gandy and published under the Creative Commons BY 3.0 license: `http://www.flaticon.com/free-icon/undo-arrow_25249`

user interface, users would have to access and change the *rdfedit* source code. Because of the lack of customization options and suitability for expert users, *rdfedit* does not comply with this heuristic. Such options will have to be implemented in the future to overcome potential usability issues for expert users.

5. Provide Users with Context. The main points Budd lists for providing the users of a web application with context do not entirely apply to *rdfedit*. Users should be aware of where they are in the application and where they came from. A common solution on modern websites are breadcrumbs and navigational sidebars that help people navigate on a website. *rdfedit* tackles this approach by highlighting the `triple-table` and especially emphasize the mouse cursor currently hovers over (cf. Figure A.2, p. 96, second row of the triple-table). Another demand proposed for this heuristic is to provide users of a web application with feedback messages. When a new triple, i.e. row is added, the table will highlight the new row green for a short period of time and jumps to that new triple. The add button simultaneously changes its color to green and its message to "Success". If the triple that should be added isn't valid, the button changes its color to red and displays an error message. When the delete icon for a triple is clicked, the according row is colored red and then removed from the table. *rdfedit* tries to give the user appropriate feedback messages to whether his operations have been successful or not and hence complies with this heuristic.

6. Consistency and Standards. While the second heuristic states that every item in the interface should be meaningful and understandable by the user, this heuristic requests developers to put the items where users would expect them to be. For example, search bars should always be found on the top part of a website. *rdfedit* has a search field that filters all items on above the `triple-table` and also has search fields for each column of the table. Latter can be found below the last table entry. What might cause an issue is that predicate cells are not editable and instead redirect users to the predicate's definition (cf. 4.3.6, p. 35). Section 5.5.6 (p. 75) discusses the reasons and implications of this circumstance. It is difficult to tell whether *rdfedit* follows this heuristic consistently. Other usability testing methods such as thinking aloud tests or observations would provide better insight on whether all items of the interface are there were users expect them to be.

7. Prevent Errors. This heuristic requires an application to limit its options in order to not fail to operate when receiving invalid data. *rdfedit* tries to follow this heuristic in multiple instances. For once, when not entering an URI in the subject or predicate field when adding a new triple, this invalid triple is not created. Second, when adding new triples, an auto-completion feature helps the user to faster input the value they desire. This does not only save time but also prevents typing errors. Third, administrators can determine query and mapping presets *rdfedit*'s users can choose from to import data from external resources. Users are not burdened with the selection of the right vocabularies and ontologies. Although *rdfedit* anticipates some errors, there are probably other ways to cause errors inside the web application. For example, right now *rdfedit* only accepts files in the RDF/XML format. When RDF graphs in other serializations are uploaded to the interface, the application fails. Section 5.5.2 (p. 72) elaborates this issue more deeply. It can't be said whether *rdfedit* complies with this heuristic. Other usability testing methods need to be applied to find other potential error sources.

8. Help users notice, understand and recover from errors. Although *rdfedit* displays an error message when an operation fails, it does not give any details about what exactly caused the operation to not succeed. To implement useful error messages that help users understand their mistakes, potential error sources need to be determined through other methods of usability testing. For now, *rdfedit* is not compliant with this heuristic.

9. Promote a pleasurable and positive user experience. Budd mentions that users should like to use a web application and it should be visually pleasing. *rdfedit* uses the Twitter Bootstrap CSS design framework[61] which is widely used across the web. According to its own usage statistics[62], Twitter Bootstrap is used by more than 3.6 million websites across the Internet. The design elements of that framework have been tested and refined in order to be visually appealing. Thanks to bootstrap, websites and web applications can easily be adapted for mobile devices. Although *rdfedit* was designed to have more visual appeal than a raw black and white table, its actual visual appeal to users can only be determined through further qualitative or quantitative usability testing. Thus it cannot be stated whether *rdfedit* complies with this heuristic.

[61] `http://getbootstrap.com/`

[62] `http://trends.builtwith.com/docinfo/Twitter-Bootstrap`

Summary. *rdfedit* was designed and implemented with the aforementioned heuristics in mind. While some circumstances that might cause unsatisfactory results have been anticipated, other sources of errors have not yet been revealed. Concerning the "Heuristics for Modern Web Application Development" by Budd [2007], *rdfedit* complies with most of them. Only by performing further usability tests it can be determined how usable *rdfedit* actually is for real users. For further examination, a thinking aloud test was conducted in order to examine whether *rdfedit* is being used as initially intended.

5.3 Thinking Aloud Test

5.3.1 Experiment Setup

Section 4.3 (p. 29) introduced the various features *rdfedit* offers, and explained how these functionalities can be used, and described what happens when they are used. These implementation steps tried to anticipate potential user behavior and thus tried to create a user friendly interface.

A thinking aloud test was prepared and conducted where the participants were asked to perform nine tasks using *rdfedit*. Four participants have been recruited to take part in this thinking aloud test. Two of them are Semantic Web novices and have never worked with Semantic Web technologies directly. The other two can be described as expert users of Semantic Web technologies since they work with related technologies on a daily basis. Each group represents a different dimension on Nielsen's user cube [Nielsen, 1994, p. 43]. All of the participants have an academic background in the (digital) humanities or social sciences and thus are part of *rdfedit*'s target audience. The participants were all directly asked whether they would like to participate in this thinking aloud test.

By examining different user groups, the diversification principle (German "Streuungsprinzip", Behnke et al. [2006, p. 195]) is applied which allows to find the commonalities and differences between the participating user groups and hence make derivations about their individual needs. The information gathered can be used to create either separate and improved user interfaces for both groups or make changes to a singular interface. When latter is the case, one should find the right compromise of simplifying a user interface for novice users while not limiting the possibilities for expert users and vice versa.

Before starting the test with each participant, they signed statements of agreements that assured them of their data being used and kept pseudonymous, as well as not

handed over to third parties (cf. Appendix C, p. 115). By pseudonymization the participants' data, their name and other identifying characteristics are replaced by an anonymous label to prevent or at least to complicate the identification of the participants. This document also informed about the setup and goals of the thinking aloud test. After agreeing, they were handed instruction sheets which stated what kind of task they should perform (cf. Appendix D p. 117).

Every test was recorded using the screencast software Kazam[63], capturing the participants' actions on the screen and the audio of their "loud thinking". Because verbatim transcriptions of the entire tests would have gone beyond the possibilities of this thesis, bullet point protocols for each test can be found in the appendix (Appendix E, p. 120). The audio-visual material recorded from the tests with participants who consented the publication of that data can be accessed on the DVD enclosed to this thesis.

For standardization purposes, all tests were conducted on the same computer, an ASUS ZenBook UX32VD laptop running Ubuntu Gnome 14.10 using the Firefox browser (version 33.0) including the option to use a wireless, optical mouse.

In the beginning, the participants were asked to perform simple tasks such as adding triples, editing and deleting them, and reversing their actions. These tasks were repeated with slightly different instructions (e.g. add a triple with another predicate: `dc:contributor` instead of `dc:creator`). The repetitions of those tasks ensured the examination of whether a learning effect occurs after the first iteration.

After finishing the simple tasks, the participants should perform more complex tasks like importing triples from external resources, performing bulk edits and initiate a literal-to-URI-conversion. Also these tasks were repeated and varied in order to detect a possible learning effect.

The thinking aloud test was concluded by a short interview where the participants could give further insight about what they liked and disliked about *rdfedit* and what they would like to see improved.

5.3.2 Hypotheses

Before conducting the test, nine hypotheses were constructed regarding the usability of *rdfedit*. The thinking aloud tests should help to evaluate the validity of these hypotheses. Each of the nine hypotheses have been tested by at least one corresponding user task throughout the thinking aloud test (cf. 117).

[63] `https://launchpad.net/kazam`

(1) **Upload:** All participants should be able to upload a RDF/XML on the *rdfedit* start page to initiate the editing process within a few seconds.

(2) **Adding:** Novices will have trouble adding triples on their first try but will easily succeed on the second iteration of this task. Expert users however will accomplish this task quicker than the novice users. All users will make use of the auto-completion feature.

(3) **Deleting:** Participants will be able to delete single triples/rows quickly.

(4) **Undo:** All participants will be able to easily undo their last actions.

(5) **Editing:** Participants of both groups will be able to make alterations to a subject or object quickly.

(6) **Bulk Editing:** Participants will need time to understand how to apply a bulk edit.

(7) **Triple Import:** Participants will have trouble to understand the workflow of the triple import and mapping process on the first iteration. Novice users might not select the most appropriate RDF graph URI to import triples from. Within the second or third iteration, this import task will be easy to accomplish by both user groups.

(8) **Literal-to-URI-Conversion:** Participants might not notice the dropdown menu appearing in the object cell of a newly added triple. Novice users will have a harder time picking an appropriate URI to replace the literal object with than expert users.

(9) **Export/Download:** All participants should be able to export the RDF graph with its alterations to a RDF/XML file.

5.4 Results of the Thinking Aloud Test

This section lists the results of the thinking aloud test. First, the results are being analyzed towards revising the hypotheses stated in section 5.3.2. The evaluation of this thinking aloud tests then closes with addressing parts of *rdfedit* that were praised and criticized by the participants.

5.4.1 Upload

The first task of the thinking aloud test asked the participants to upload a provided RDF/XML file to *rdfedit*. All participants immediately navigated to the file selection (Choose) button on the start page (cf. Figure A.1, p. 95) and selected the appropriate file.

Afterwards, three out of four participants pressed the Upload button to execute the file upload. The fourth participant however expected the RDF/XML file to be immediately processed on selection in the file browser, hence voiding the hypothesis of all participants being able to upload the file quickly. Expecting instantaneous processing is understandable, since that kind of behavior is common on many websites (e.g. Dropbox[64] or Facebook[65]). To add to the responsiveness of *rdfedit* and to better comply with the consistency and standard heuristic proposed by Budd [2007] (cf. p. 57), the file upload process has likely be reduced from two steps – Choose File & Upload – to a single action, or even offer drag and drop functionalities.

5.4.2 Adding

After having uploaded the graph file, the participants got to see the tabular RDF editing interface of *rdfedit* for the first time. They were asked to add three triples to the table. After a quick examination of the whole interface, three of four participants found the input fields to add new triples within a few seconds (cf. Figure A.3, p. 96, marked blue). The last participant took about 15 seconds to find these fields.

To represent the functionality of these input fields, they display a place holder value (e.g. `Add Subject`) that vanishes when users click into the field to enter new values. This indication does not seem to be sufficient, since all participants of the thinking aloud test loudly wished for a better emphasis of the whole add triple input fields and button group, for example by adding an additional heading to that row. After they had used the input fields and add button for the first iteration of this task, the participants became quicker during the second and third triple addition.

After the first iteration, the participants encountered a bug in the software that has actually been noticed prior the test and apparently falsely assumed to have been fixed. The place holder values of the input field stopped automatically vanishing, when click-

[64] `www.dropbox.com`
[65] `www.facebook.com`

ing into the fields. Users had to select the whole place holder text and delete it to enter a new value.

While adding triples all participants made heavy use of the auto-completion feature discussed in section 4.3.3. For almost every subject, predicate and object, the participants only typed a substring (e.g `ore:a` instead of `ore:aggregates`) and then chose the appropriate URI by navigation through the list of suggestions with the arrow keys and then confirmed by pressing the tabulator key, or they selected that URI using the mouse cursor. One novice participant remarked, that she could not identify a sorting in the auto-completion suggestion list and proposed to sort the suggested values first alphabetically and secondly by their string length to allow a better overview and thus a quicker access to the right URI.

It appears that the amount of Semantic Web background knowledge did not matter in this task at all. The speed of adding new triples rather depended on the participants' preference of navigating dropdown menus. The two participants, one expert and one novice, mainly using the arrow keys were faster than the other two who preferred the mouse.

The lessons learned from this task is to add better indications of actions, e.g. by giving headlines or separating those actions more from the triple-table, to fix the input field indicator bug and to sort the auto-completion suggestion values. This shows that the clarity heuristic evaluated earlier (cf. p. 56) has not been applied as thorough as initially assumed. However, the efficiency and task completion heuristic seems to have been abided.

The results of the triple adding task only semi-validated the corresponding hypothesis. While the assumptions of a learning effect occurs while using that feature repeatedly and the utilization of auto-completion by all participants have been proven as correct, the supposition of Semantic Web experience influencing the speed of accomplishing the given tasks turned out to be wrong. Actually, background knowledge in Semantic Web technologies not mattering can be viewed as a positive trait of *rdfedit*, since potentially users of all experience levels can use the adding feature.

5.4.3 Deleting

In the third task, the participants were asked to delete three triples from the triple-table. Two of these triples were predefined by the instructions while the last one only needed to have `dc:language` as the predicate. Implicitly, this task tested if the participants were able to identify the delete icon as such. Apart from testing whether users can

actually delete triples using *rdfedit*, the task design also intended to examine whether participants make use of the table-wide search or column filter functionalities described in section 4.3.4.

As it turns out, all participants made use of the column-filter boxes (cf. Figure A.6, p. 97), often even using more then one simultaneously to find the correct triple to delete. All participants stated that they found the possibility to search for particular subjects, predicates and objects very helpful.

When they had identified the right triple to delete, they clicked on the delete icon on the right side of the particular row. Two participants liked that the deleted triple is highlighted in red for a few seconds before it disappears. However, the participants also made remarks on how they would like to see this feature be improved.

For once, an additional confirmation-popup asking whether a triple should really be deleted in order to prevent accidental deletions would be helpful. This idea would greatly contribute to Budd's heuristic of preventing errors and providing a more pleasurable user experience (cf. p. 57). However, that confirmation feature would needed to be tested separately. The current web applications (e.g. Google Mail or Dropbox) rather delete data instantly and offer a dedicated undo mechanism to restore that particular data instead of requesting the users' confirmation. If the participant's suggestion would be implemented, users would receive confirmation requests for every triple that should be deleted. This method would require more interaction between the web application and its users and thus would slow down the process of editing data with *rdfedit*. Offering dedicated restore functions decreases the numbers of user interactions, making users only need to restore accidentally deleted triples and thereby increasing the speed of editing RDF data.

The second suggestion was to display a tooltip message showing the meaning of the delete icon in text form when the mouse cursor hovers over and thus making the user interface clearer to the user.

Still, the hypothesis of all participants being instantly able to delete triples turned out to be correct.

5.4.4 Undo

This task asked the participants to revert the deletions made in the previous assignment. The main intentions behind this task were to find out how fast the participants were able to find a method to undo the changes and what method that was. Section 4.3.9

presented two options to revert operations performed within *rdfedit*: an undo button and an undo action in the top bar.

One of each user group used the undo button on the right bottom corner of *rdfedit* whereas the other two participants accessed the action via the top bar. The participants using the undo button took longer to find it than the others using the top bar undo option. One participant of the latter group stated, that the undo action was right there, where she expected it to be, and thus confirming a statement made by one of the button users. She expected the button to be above the table, not below it.

Moreover, one of the participants asked whether there is a `Ctrl+Z` keyboard shortcut that does revert the last change, as it does in many other applications. Implementing that keyboard shortcut would likely provide a standardized way how to revert actions within *rdfedit*.

During that test, another bug in the web application was uncovered. While the undo button reverted actions flawlessly, the undo option inside the top bar only worked for a single reversion. Nevertheless this bug did not interfere with the implicit question on how fast they were able to find ways to reverse actions within *rdfedit*.

Still the executions of the undo task have shown that the participants were not able to find the undo button easily, hence disproving the assumption made for this task. To fix this issue the undo button should be moved to above the triple-table.

5.4.5 Editing

The simple editing feature (cf. section 4.3.5, p. 33) was examined through letting the participants edit three objects of triples with a given value.

One novice participant did not understand what to exactly do since, as she stated, she did not completely grasp the concept of the subject-predicate-object model yet. After re-reading the introductory page of the test instructions (cf. p. 117) the task became clearer.

To find appropriate triples, all participants used the object-column-filter. Three participants simply clicked onto the object-cell which invoked the object-values to become editable. One expert user however looked for a dedicated edit button until stumbling upon the possibility to click onto an object-cell.

Although the participants could edit the object values, they were not satisfied with how the edit is being initialized. As for now, clicking onto a cell activates the editing mode of a cell, and clicking somewhere else ends that editing mode. A bug caused the editing box to also close when another click was made on the same object-cell. Hence

trying to activate the editing box with a double click or wanting to reposition the text cursor aborted the editing mode.

Two users wanted to apply the changes they made with the enter key. Because the edit boxes are HTML5 `contentEditable` nodes, pressing enter causes a line break inside that edit box. Using the enter key has not been anticipated while implementing *rdfedit*. One expert user figured out to only click once and then use only the keyboard to make the alterations and then click somewhere else to apply the change. She suggested to make the table behave more like Microsoft Excel, i.e. editing a cell only on a double click. That suggestions goes along with Budd's first heuristic (cf. 55) of anticipating user expectations and making the web service feel like an "offline application".

In the end, the hypothesis made for editing triples with *rdfedit* turns out to be false. The participants conceptually grasped how to edit triples quickly, but failed since *rdfedit* did not behave as they expected. Issues like double clicking not starting an edit mode or confirming changes with the enter key might cause frustration while using the web application and will be fixed shortly.

5.4.6 Bulk Editing

To test how usable the bulk-editing feature (cf. section 4.3.10) is, the participants were presented with two consecutive assignments. The first task asked them to alter a specific subject value that appeared multiple times inside the RDF graph. The follow-up task instructed them to apply that change to all other triples with the same value, without specifying how.

After having learned how to change objects, all participants instantly knew how to apply changes to a single subject. When a subject is changed, the bulk-edit icon appears in the cell of the same subject. When that icon is clicked on, *rdfedit* executes the bulk-edit and changes all remaining triples with the same value.

It appeared that the bulk-edit icon ( ) has not been the right choice to represent that operation. Although all participants recognized the icon appearing after they had changed a subject, everybody assumed it's purpose is to undo the subject change. In other web contexts, the bulk-edit icon indicates a "reply to all" operation. *rdfedit* unsuccessfully tried to borrow and slightly alter that meaning to "apply to all". When examining pictrograms web-icon-sets like Fontawesome[66] have to offer, there is no particular icon clearly giving an "apply to all"-meaning.

[66] `http://fontawesome.io/`

Only one participant gave the icon a try and was pleasantly surprised about the bulk-edit and called this feature very helpful. The remaining three participants had to abort that task and then were shown what the bulk-edit icon does. Two of them also stated that they liked this feature but that icon's purpose wasn't clear to them. One novice user tried to mark access cells at once "like in Excel" to apply a bulk-change.

Meanwhile testing, a bug in the bulk-edit feature appeared. When the bulk-edit icon was clicked, the bulk-edit was carried out but the subject value the edit originated from was changed back to its original value. Still, this bug is rather an issue of functionality than usability, since the main goal of the two tests was to estimate whether the participants were able to find and use the option to perform bulk-edits.

The hypothesis about the participants being able to easily apply a bulk-edit failed. To improve the usability and understandability of the bulk-edit feature, other icons need to be tested. Moreover, thanks to one expert participant's suggestion, the bulk-edit icon should also receive a tooltip to provide other users with feedback on what that icon exactly does.

5.4.7 Triple Import

To test whether people can use the triple import and mapping feature, the participants were given query keywords and corresponding query-preset labels (e.g. `Herman Melville` and `Person`) and then asked to import triples (cf. section 4.3.12, p. 43).

Three participants instantly found the input field for the query keywords and the dropdown-menu where they can set the according type (cf. Figure A.11, p. 100). One novice user needed more than 15 seconds to find the necessary elements inside *rdfedit*'s tabular interface.

All of the participants entered the query keywords and set a type using the dropdown-menu. Afterwards, only three of the users clicked on fetch graphs button to let *rdfedit* query Sindice and look for relevant URIs for the entered query keywords. One expert participant did not expect that kind of workflow and anticipated that the web application would fetch the URIs after she had chosen a type.

As soon Sindice retrieved the URIs, all users noticed that the choose graph dropdown-menu had changed and now contained new values. All users always chose an appropriate URI to accomplish this task. The second and third iteration of this task was easily managed by the participants.

Unfortunately, DBpedia could not be accessed while performing the thinking aloud test with one participant. The `Person`-query-preset demands *rdfedit* to retrieve triples

from the DBpedia. However, *rdfedit* was still able to fetch relevant graph URIs since Sindice has some DBpedia data indexed. When that user decided for an URI, the *rdfedit* could not load the corresponding graph and thus not import any triples. Nevertheless, the task was still carried out as far as possible to see whether the participant was able to operate the triple fetching feature.

The third iteration of this task asked the users to query for the location `Berlin`. The `location`-query-presets limits the results to graphs from geonames.org. Unfortunately, geonames does not use "talking URIs". Instead of something like `geo:Berlin` they used numeric identifiers[67], which made it hard to choose the most fitting URI.

One expert user suggested instead of displaying the graph URI addresses, to extract the human-readable labels from those graphs (e.g. `Berlin, Germany`) and present them for selection. This idea is worth considering. Since administrators of a *rdfedit* need to have some background knowledge about the data models in specific domains anyway in order to construct the query- and mapping configurations, all they needed to add to those configuration files is where the most human-readable labels lie in the graphs of the respective databases. By exclusively displaying human-readable labels and "talking URIs", choosing the most appropriate RDF graph would probably become easier for other Semantic Web novices.

Furthermore, another participant suggested to also emphasize the input field and buttons more heavily so other users might find them quicker.

The hypothesis that both participant groups were able to use the triple import feature with ease during the second and third iteration turned out as correct. While this feature all in all seems to be usable, this task determined some minor usability issues, which, when fixed, probably provide a better user experience.

5.4.8 Literal-to-URI-Conversion

The second to last task again let the participants add triples manually in order to invoke the literal-to-URI-conversion feature discussed in section 4.3.14. All participants again made heavy use of the auto-completion feature in order to add subject and predicate values. Since the instructions only proposed literal object values that did not yet occur within the graph, these values were not suggested by the auto-completion feature.

As soon the triples were added, all users noticed the dropdown-menu appearing next to the new literal value inside the object cell and immediately chose one of the

[67] For example: `http://sws.geonames.org/2950159/`

suggested URIs to replace the literals with. All participants later stated that they found this feature to be very helpful.

Only one participant encountered a minor problem. When using the search options, the triple-table only displays values matching the keywords inside the search boxes. When new triples are being added, that filter still applies and users cannot see whether a triple has been successfully added or not. To improve user feedback, the search boxes will be reprogrammed to clear themselves as soon as a new triple gets added to the table.

The hypothesis for this task pessimistically assumed that users would not notice the option that literals could be semi-automatically replaced by URIs. It turns out that this feature is better usable than expected and thereby voiding the hypothesis that was phrased for this task. Moreover, both novices and experts could use this feature with ease.

5.4.9 Export/Download

In the last task of the thinking aloud tests the participants were asked to download the edited RDF graph to the hard drive.

Both expert participants either located the option to export the graph in the top bar or the export button in the bottom left corner of the interface almost instantly. The third participant needed a little longer and decided to use the top bar export option. The last novice participant did find the option but did not understand what it means. The button and the top bar options are labeled with `Parse to RDF`. She had to ask what "parsing" means in order to understand what the button does. Only after an explanation about that term the participant was able to accomplish the task, hence voiding the hypothesis that all participants should have been able to export the graph to a RDF/XML file.

Throughout the tests with the participants, three of them stated that they preferred a more concise label like "export" or "save". Since the label as it is now raised too many concerns, it will definitely be changed shortly. One participant proposed to enable the export to different file formats. That issue is later discussed in section 5.5.1 (p. 71).

5.4.10 Summary

All in all, the participants of the thinking aloud test were able to successfully accomplish all given tasks, except the assignment focusing on bulk-edit and exporting the changed

graph. The main causes for these failures were an unclear icon choice and unintelligible labeling.

Four out of nine hypotheses turned out to have made wrong assumptions. Minor usability issues caused at least one participant to take longer to accomplish a task than expected or not being able to complete a task after all. One novice clearly had more trouble completing the tasks than the other three participants. That participant, as she stated, did not completely grasp the idea behind the Semantic Web and the RDF syntax, but was still able to complete almost every task. The thinking aloud test has shown that rdfedit does not comply with some the heuristics discussed in section 5.2 which have been assumed to be met prior to the usability test.

Five assumptions have been proven as correct. Unexpectedly, the participants had less trouble operating *rdfedit*'s more complex features like triple-importing and literal-to-URI-conversion. All participants praised that *rdfedit* is fast and easy to operate and that they are especially liking the search, auto-completion, import and conversion functionalities of the software. Moreover, they found the bulk-edit feature very helpful as soon they either have been shown or discovered it themselves. Their appraisal corresponds to the goals defined prior to *rdfedit*'s implementation and testing (cf. Table 3.1, p. 13), thus indicating *rdfedit* and its semi-automatic features really help aid, even Semantic Web newcomers, in the generation of RDF data.

The research question, whether people can create RDF data in a short period of time without needing to know much about Semantic Web technologies can be answered with a positive tendency. Since only two novice users are not a representative number of testers, *rdfedit*'s usability has to be examined with a greater number of people. Nevertheless, the results of the thinking aloud test indicate that *rdfedit* can be used by people with no Semantic Web background knowledge, enabling them to create, manipulate and aggregate RDF data.

5.5 Self-Criticism

The heuristics (section 5.2) and the thinking aloud test (section 5.3) could only evaluate the interface visible to users; the back-end of *rdfedit* remains invisible to them. Hence, participants of the thinking aloud tests could not make any statements about the internal performance or the conceptual complexity of the features available in *rdfedit*. The next pages focus on issues that retrospectively appeared throughout the development

process of *rdfedit* and features that were once considered to be added to the application but did not make it into this version of *rdfedit*.

5.5.1 Back-end Format

In the early stages of *rdfedit*'s development, various RDF serializations that may deem viable as a back-end format for the tabular triple editing interface have been glanced over. String-based serializations such as `N3` or `Turtle` were never considered because it is harder to manipulate them in an automatic and structured manner. More generally applied formats like XML and JSON offer an easy and direct access to their already structured content.

Although RDF/XML is one of the heavily used RDF serializations to store RDF data, the complexity of implementing RDF/XML manipulating functions inside Java-Script would have been much higher compared to the JavaScript-native JSON format. Hence a JSON-based RDF serialization should be used for *rdfedit*.

During the search for an ideal format, RDF/JSON was evaluated as a good choice because of its structural simplicity and thus easy accessibility for various programming languages. Since the early code-base was implemented using that format, another JSON-based serialization was overlooked in the early stages: JSON Linked Data (JSON-LD) [Sporny et al., 2014].

While RDF/JSON remains a W3C Working Group Note since the end of 2013, JSON-LD has reached the high status of being a W3C recommendation. Following discussions on the CODE4LIB and W3C Semantic Web mailing-lists throughout this year, it seems there also is a higher interest and application rate of JSON-LD than RDF/JSON.

Apart from its popularity, JSON-LD holds other advantages over RDF/JSON. JSON-LD is a more general serialization for Linked Data and can also be used to express Microdata [Hickson, 2013] and Microformats[68], both heavily used by major search engines and other (commercially) successful websites.

At this stage of development, switching switching the back-end RDF format is strongly considered. Recently, RDFLib was extended to also work with JSON-LD. Moreover, jQuery libraries have been published that focus on the manipulation of JSON-LD inside the browser.[69] In general, new JSON-LD modules and libraries were developed for all major programming languages. On the contrary, all JavaScript and jQuery functions to

[68] `http://microformats.org/`

[69] `https://github.com/digitalbazaar/jsonld.js`

manipulate RDF/JSON data inside *rdfedit* had to be manually written, since there were no libraries available that could have been reused.

Switching over to JSON-LD would not only simplify the development process and creation of forks of *rdfedit*, it would also create the possibility to extend *rdfedit*'s feature set to read, manipulate and export other types of Linked Data. Hence, a substitution of RDF/JSON by JSON-LD in future versions of *rdfedit* will be very likely performed.

5.5.2 Format Compatibility

At the time being, *rdfedit* is only compatible to one in- and output RDF serialization: RDF/XML. Another format, RDF/JSON also works with *rdfedit* but only as an intermediary to transmit RDF graphs between the users' computers and the *rdfedit* server.

Although the RDF/XML notation is very complex and thus not ideal for manual notation [Bray, 2003] it still is a widely used serialization of RDF that is compatible with the majority of Semantic Web related software. Hence concerning the variety of RDF serializations, *rdfedit* should focus on being able to read and export RDF/XML.

RDF/JSON on the other hand is fairly easy to notate but lacks the expressiveness that can be achieved by using other RDF notations [Dodds, 2010]. Simultaneously, because it is a JSON based-format it is very easy to handle within a JavaScript environment.

By using RDFLib to analyze graphs uploaded by users, it is possible to read and write different RDF serializations (e.g. RDF/XML, Turtle, JSON-LD) within the Django ecosystem. Hitherto, the support for other input formats than RDF/XML has not been implemented because the implementation of such feature would have transcended the scope of this thesis.

It is possible to check the MIME-type of a graph-file and thus choose the correct RDF parser for the serialization. If the MIME-type is wrong or absent, *rdfedit* could rely on choosing the appropriate way to read the graph by its file extension. For example, a file with the MIME-type `application/rdf+xml` or with the file extension `.rdf` would instantly be regarded as a RDF/XML file. However, if there is a different serialization used than indicated by the metadata or the filename contains an unknown extension, parsing the graph would fail. The only proper solution on how to interpret different RDF file formats correctly is to analyze the file content's syntax. Since every serialization has its own unique syntax, *rdfedit* would have to match the file contents against regular expression patterns and then validate whether the syntax was used correctly.

Since the focus of this thesis is to implement a web-application that aids people with the generation of RDF data in a semi-automatic way, ensuring compatibility with a

wide range of RDF formats would have been a too time-consuming task that would not have added too much to the user-aiding picture. Nevertheless this is an issue worthy to be investigated further since supporting more file formats also means being able to broaden the potential user-base of *rdfedit*.

5.5.3 Reversing Actions

During the presentation of the triple-import and mapping feature as well as the literal-to-URI conversion problems that occur during the reversal of these actions were mentioned (cf. section 4.3.13, p. 48 and section 4.3.14, p. 50).

When triples are being imported from external resources, they are added one by one to the graph leaving the user to press the undo button multiple times to get rid of all imported triples. Since *rdfedit* lacks an indication on what action was performed last, e.g. having added a specific triple, it is safer to delete these triples manually rather just "undoing" them. By clicking on the undo button repeatedly there is the chance of clicking too often and thus reverting actions that should not have been reverted. This issue presents a major concern regarding the user-friendliness of *rdfedit* and thus will be worked on in the near future.

A similar issue arises when converting literal objects to URIs. So far, a conversion can only be performed when a new triple having a literal object was added to the graph. Since *rdfedit* uses its triple adding function to implement data from external RDF graphs, the conversion is both possible for manually and semi-automatically added triples. As soon as users decide what URI to substitute the literal object with, the dropdown menu users can choose suggested URIs from is removed from the object cell in the user interface. If the undo button is pressed afterwards, the object is changed back to is literal value but not showing the URI dropdown menu anymore, i.e. it is not instantly possible to choose another URI for the conversion process. Instead, the whole triple had to be deleted and reentered in order to request the URIs again and display the dropdown menu for the user. This issue also flaws the user-friendliness of *rdfedit* because it punishes users for making a mistake by forcing them to repeat their tasks until they have chosen a URI they are confident with. Since there is a potential of frustration when reversing a literal-to-URI-conversion, eliminating this issue is a high priority within the next steps in the development of *rdfedit*.

5.5.4 User Management

As mentioned in the elaboration of exporting RDF graphs from the *rdfedit* user interface (cf. section 4.3.15, p. 51), there is currently no way for users to save their graphs on the server side of the web-application. However, implementing such a feature is possible. For demonstration purposes there is an example button on the main page of *rdfedit* (cf. Figure A.1, p. 95). When being clicked on, the application reads an RDF/XML file that is being stored on the server and processes it in order to generate the tabular user interface for that graph.

As for now, edited graphs would have to be exported to the hard drive of the users' computers and later uploaded again to progress editing them. For implementing the main goals of this application mentioned in section 3.2 (p. 12) also creating user management system and testing its functionality and security would have gone beyond the scope and possibilities of this thesis. After implementing it, one needed to ensure that only users with the right authorization can save and access their files.

Simultaneously, it had to be tested what happens when multiple users at once perform read and write operations on the server. Factors like the quality of Internet connectivity, size of the RDF graphs being worked, number of users using the same *rdfedit* at the same time and the computational power of the server and the users' machines would need to be taken into account. Only by testing scenarios where the values of those factors are varied could lead to an answer on how stable and secure a *rdfedit* server is under heavy use. Since an user management feature is absent in *rdfedit*, there is no need to worry about account data security and less concerns regarding the server's stability.

While adding user management to *rdfedit* is an attainable task, it has to be postponed and will likely be implemented and thoroughly tested at a later point.

5.5.5 No Triple Store Interface

Another issue related to not being able to store data on the server is not providing an interface between *rdfedit* and triple stores. For productive, RDF-extensive work environments where new data should be ingested by a triple store on a regular basis, it would be convenient to transfer triples created with *rdfedit* directly into the database. This wish was expressed by Semantic Web experts (other people than the thinking aloud test participants) who suggested such kind of feature to quicken the addition of new triples to triple stores.

For this thesis however, creating and testing interplays between various triple store software products and *rdfedit* would have been an effort too great to accomplish. Although SPARQL recently was extended by the ability of directly writing into the respective triple stores behind the SPARQL endpoint being queried [Arnada et al., 2013], finding the most effective method to insert and update triples inside a triple store would still have required testing different scenarios (amounts of triples, correct order of insertions, avoiding inconsistencies).

While *rdfedit* tries to simplify the process of creating RDF data for Semantic Web newcomers, that doesn't mean the data these newcomers produce are correct or consistent. Just altering a single triple inside a triple store could cause a small inconsistency which might lead the databases internal reasoning software to dismiss some triples since they no longer match a particular schema or ontology.

If *rdfedit* is to be used in a productive environment, the edited graphs need to be exported into a file and forwarded to the triple store managers who confirm the validity of the data and upload it into their knowledge base. Hence, a more reasonable way of providing some kind of connection to triple stores is to build upon the user management system described earlier (section 5.5.4). If *rdfedit*' users could simply tag graphs as ready as soon as they have finished working on their data, the triple store managers could just take these graphs from the *rdfedit* database, examine them and insert those graphs into their triple store. Similar to the user management system, implementing this idea is a task for the future.

5.5.6 Predicate Editing

While editing the subjects and objects of single triples has been thoroughly described in section 4.3.5 (p. 33), editing predicates has been left out so far. Although the implementation of predicate editing is just as possible as changing the rest of the triples, it was a conscious decision to not include predicate editing within *rdfedit*'s feature frame.

Because of the nature of the Semantic Web it is very likely that subject and object values are more varied than predicates. Predicates are more likely to originate from an already existing vocabulary with clear definitions when a certain predicate value should be used. If *rdfedit* is being used as initially intended, users create instance metadata using the web application, i.e. creating new, not yet existing name spaces for the resources they want to describe and applying those to subject and object URIs while they add predicates using well established vocabularies, such as Dublin Core, FOAF etc. The reuse of exiting vocabularies is recommended and supported by *rdfedit* since

the users, who are Semantic Web newcomers, should not be burdened with the creation and testing of their own vocabularies and *rdfedit* supplies common vocabularies by auto-completion during the addition of new triples.

As laid out in section 3.1 (p. 12), the responsibility of creating RDF data is split among three parties: the users themselves, administrators of the web application and *rdfedit*. When configuring *rdfedit* administrators can choose which vocabularies to include for the auto-completion process and thus limit the users' possibilities of choosing predicates outside the predefined vocabulary set. In the ideal case, users should only need to choose among vocabulary implemented and not need to refer to others. Hence, the chance of creating a triple with a "valid" URI is very high. Clicking on the predicate field on purpose or by accident would only increase the chance to insert a typo and thus generate errors inside the RDF graph.

However, this solution is also not very ideal since it also reprimands the user for wanting to change a predicate. The state of *rdfedit* now requires the triple with the wrong predicate to be deleted and re-added with the correct URI. This issue can be fixed by allowing users to only choose from a set of predicate URIs. Since it might be the case when other vocabularies have to be used, such URIs could be added too but had to be confirmed in order to make the users check whether they entered a valid value.

5.5.7 Dependencies & Version Compatibility

Starting this web application requires the version of at least Django 1.5.1. At the beginning of *rdfedit*'s first implementation steps, that version 1.5.1 was the most recent one available. For *rdfedit* to properly run with Django 1.5.1 some Python libraries need to be installed on the system hosting the web application:

(a) **Dajaxice**: A Python/Django library that enables AJAX functionality inside Django web applications. This library is crucial for the triple-fetching, triple-mapping and literal-to-URI-conversion features of *rdfedit*.

(b) **RDFLib**: RDFLib is a Python library to process RDF data (cf. section 4.1.2, p. 24 for more details) and is necessary to process uploaded graphs, extract triples from external graphs and export edited graphs to the users' hard-drives.

(c) **RDFLib-jsonld**[70]: This RDFLib extension ensures compatibility with JSON-LD. Although *rdfedit* doesn't support JSON-LD yet, the utilization of that format is a striving goal within the further development of *rdfedit*.

(d) **RDFLib-sparql**[71]: This RDFLib extension enables SPARQL capabilities within a Python environment. It is needed to transform an uploaded RDF graph to a list of triples which the tabular user interface is generated from. Additionally it is needed to extract triples from fetched graphs.

(e) **RDFLib-rdfjson**[72]: This RDFLib extension allows Python/Django to read RDF graphs in the RDF/JSON serialization. It is needed to create the RDF/JSON object which lies in the background of the *rdfedit* user interface and undergoes the changes made to the graph inside the tabular user interface. It is also needed to re-read the edited graph when the users' want to export their changed graphs. In that case, the RDF/JSON graph is sent to the server, interpreted by RDFLib and transformed into an RDF/XML file (cf. section 4.3.15, p. 51).

(f) **Widget-Tweaks**[73]: This library is needed to properly generate the HTML code for the triple-table.

(g) **simplejson**[74]: This library is needed to add basic JSON support to Python. It is used to "dump" the RDF/JSON serialized RDFLib object to proper JSON.

If one of the libraries listed is not installed it will cause an error during the startup of *rdfedit* and abort the application. Another issue observed is that *rdfedit* does not seem to run on newer version of Django. By now, Django has reached version 1.7.1 and has experienced some changes regarding its own structure and underlying model.[75] Since these changes render Django applications using older Django version incompatible with their newer counterparts, the source code of the web applications has to be migrated. Since *rdfedit* runs fine in Django 1.5.1 and has been tested in that environment, the migration has not been executed in order to stay in the scope of this thesis.

To overcome this issue it would be best to provide an installer script that downloads and installs all required libraries and the correct version of Django to run *rdfedit*. However, this would require users who want to host a *rdfedit* instance to have administrator

[70] `https://github.com/RDFLib/rdflib-jsonld`
[71] `https://github.com/RDFLib/rdflib-sparql`
[72] `https://github.com/RDFLib/rdflib-rdfjson`
[73] `https://pypi.python.org/pypi/django-widget-tweaks`
[74] `https://pypi.python.org/pypi/simplejson/`
[75] `https://docs.djangoproject.com/en/dev/topics/migrations/`

rights on their machines in order to install new software. Moreover, distributing such installers would also take a lot of time in testing since it should work on different version of Microsoft Windows, Mac OSX and various flavors of Linux/UNIX distributions such as Ubuntu, Fedora and Arch Linux. This solution will be addressed shortly after the more crucial problem mentioned in the subsections before have been fixed.

5.5.8 Namespace Manager

When using *rdfedit* most URIs are abbreviated for better readability. Additionally, short URIs are also faster to type. The settings file of *rdfedit* contains a dictionary object, which holds the namespace abbreviations and their long counterparts. For example, `http://purl.org/dc/elements/1.1/` is reduced to `dc:`. As for now, that dictionary contains the most commonly used namespace declarations.

If users of *rdfedit* wanted to add additional namespace-abbreviations or change them to match their preferences, they would need to access the settings file and make the changes themselves. This would require the users to be able to actually access the server which rdfedit is running on and editing some Python code. A better solution would be to have a namespace manager integrated inside *rdfedit*'s user interface, where users can edit namespace definitions. That manager would also provide an easier look-up to resolve the abbreviations.

Although this feature would have increased the value of *rdfedit*'s usability, it was not added to the web application in order to correctly implement the features described throughout section 4.3. A namespace manager is a feature that is considered to be added to *rdfedit* in the near future.

5.5.9 Sindice's End of Support

During the evaluation of services that can extend *rdfedit*'s functionality, the choice was made in favor of Sindice (cf. section 4.3.11, p. 39). As Franzon [2014] stated, Sindice will no longer be supported by the end of 2014. Nonetheless, some features were implemented that rely on Sindice for short term demonstration purposes of user-aiding features like importing and mapping triples from external resources.

However, building such features upon Sindice is not a sustainable solution. The most viable and long-lasting approach would be to entirely use SPARQL for querying external triples stores and omitting third party services like Sindice or Watson. Shekarpour et al. [2013b] have shown that creating SPARQL queries from templates also delivers triples

and graphs with high relevance. Implementing and testing a flexible template-based solution would have gone beyond the scope of this thesis. Executing SPARQL queries to multiple triple stores at once to obtain more possibly relevant triples and joining the results into a single set of triples is a complicated and sophisticated process [Hartig et al., 2009]. Combining both approaches, template-bases SPARQL queries and multi-triple-store-queries would be interesting material for another thesis.

Since the termination of Sindice's support is an immanent problem for *rdfedit*'s operationability, implementing a longer-lasting and independent SPARQL-based solution for retrieving triples from other databases is a high priority. If there will appear another Semantic Web search engine that can be integrated similarly easy as Sindice within *rdfedit*, that service might be favored over SPARQL-based approaches. An API-based service which preprocesses data is often less complicated to use and interact with than gathering and processing the "raw" data yourself.

5.5.10 Summary & Outlook

To make *rdfedit* suitable for everyday and institutional use, its abilities need to be extended. Adding a user management module that allows users to store graphs they are editing on the server, implementing a more direct connection to triple stores and having a *rdfedit* installer are features that could be attractive for larger organizations where RDF data is being generated.

Furthermore, the user interface still needs some improvement. Apart from the problems described in the thinking aloud test results, allowing predicate editing and enhancing the undo processes are issues that need to be tackled. A namespace manager would also provide more control and clarity to users.

Since Sindice will not remain to be a reliable intermediary between *rdfedit* and the Semantic Web, alternatives have to be evaluated. Besides that, a support of a wider variety of RDF serializations would be desirable for this web application.

However, all these points were not realizable in the scope of thesis but are likely to be tackled in the future. The usability tests have shown that *rdfedit* can be easily used by both Semantic Web experts and novices. It would be interesting to further investigate the commonalities and differences between both groups in order to create user interfaces that suit the need and skills of the respective groups.

6 Conclusion

The Semantic Web has still a long way to go in order to become a mainstream phenomenon. Although the numbers of triple stores, triples and Semantic Web users is constantly increasing, the awareness about the Semantic Web is not high enough for it to be considered as "Web 3.0".

rdfedit was conceptualized and implemented for Semantic Web newcomers in order to provide a simplified access to creating RDF instance data. The intention behind *rdfedit* is to aid users in the creation, manipulation and aggregation of *RDF* data using semi-automatic features like importing triples from external resources and converting literal objects to URIs. Instead of complicated syntaxes, visualized graph models or Wiki-like structures, *rdfedit* offers a simple subject-predicate-object table. The thinking aloud usability test showed that participants using *rdfedit* could easily pick it up and quickly add, edit and delete triples as well as import data.

Looking back at the goals defined for *rdfedit* in section 3.2 (cf. Table 3.1, p. 13) before the implementation and usability-testing of the web application, all those goals have been met by the web-application. The participants really liked the auto-completion, search, import and conversion functionalities about *rdfedit*. They were all able to create, edit and aggregate RDF data in a short period of time using the semi-automatic features provided. *rdfedit* is a little stepping stone when it comes to making Semantic Web technologies more popular and accessible.

One novice of the thinking aloud test still had some minor problems when using *rdfedit*'s interface because the concept of the Semantic Web was not quite clear. This shows that even a usable software tailored to novices users has its usability-limits, since some prior knowledge about that domain is necessary to operate that software.

This thesis has demonstrated that developing a usable and user-friendly RDF editing web application is possible. However, apart from providing a usable software people need to be educated about the underlying Semantic Web technologies at least to a certain degree. Having usable software like *rdfedit* providing an easy access to the Semantic Web, now disciplines like information-, education and social sciences need to cooperate on developing concepts to educate people about the Semantic Web. Only by increasing the public knowledge and the application rate of Semantic Web technologies we might be able to realize the vision of semantic agents.

References

[Adida et al. 2012] ADIDA, Ben ; HERMAN, Ivan ; SPORNY, Manu ; BIRBECK, Mark: RDFa 1.1 Primer: Rich Structured Data Markup for Web Document / W3C. Version: Juli 2012. http://www.w3.org/TR/2012/NOTE-rdfa-primer-20120607/. 2012. – Working Group Note

[Antoniou and Harmelen 2008] ANTONIOU, Grigoris ; HARMELEN, Frank v.: *A semantic Web primer*. 2. ed. Cambridge, Massua : MIT Press, 2008. – ISBN 9780262012423

[Arnada et al. 2013] ARNADA, Carlos B. ; CORBY, Olivier ; DAS, Souripriya ; FEIGENBAUM, Lee ; GEARON, Paul ; GLIMM, Birte ; HAWKE, Sandro ; HARRIS, Steve ; HERMAN, Ivan ; HUMFREY, Nicholas ; MICHAELIS, Nico ; OGBUJI, Chimezie ; PERRY, Matthew ; PASSANT, Alexandre ; POLLERES, Axel ; PRUD'HOMMEAUX, Eric ; SEABORNE, Andy ; WILLIAMS, Gregory T.: SPARQL 1.1 Overview: W3C Recommendation 21 March 2013 / W3C. Version: März 2013. http://www.w3.org/TR/sparql11-overview/. 2013. – Report

[Arvidsson et al. 2012] ARVIDSSON, Erik ; SMITH, Michael ; BARTH, Adam: URL: W3C Working Draft 24 May 2012 / W3C. Version: Mai 2012. http://www.w3.org/TR/2012/WD-url-20120524/. 2012. – Report

[Auer et al. 2006] AUER, Sören ; DIETZOLD, Sebastian ; RIECHERT, Thomas: OntoWiki textendash A Tool for Social, Semantic Collaboration. Version: Januar 2006. http://link.springer.com/chapter/10.1007/11926078_53. In: CRUZ, Isabel (Ed.) ; DECKER, Stefan (Ed.) ; ALLEMANG, Dean (Ed.) ; PREIST, Chris (Ed.) ; SCHWABE, Daniel (Ed.) ; MIKA, Peter (Ed.) ; USCHOLD, Mike (Ed.) ; AROYO, Lora M. (Ed.): *The Semantic Web - ISWC 2006*. Springer Berlin Heidelberg, Januar 2006 (Lecture Notes in Computer Science 4273). – ISBN 978–3–540–49029–6, 978–3–540–49055–5, 736–749

[Beckett and Berners-Lee 2008] BECKETT, David ; BERNERS-LEE, Tim: Turtle-terse RDF triple language: W3C Team Submission 28 March 2011 / W3C. 2008. – Report

[Behnke et al. 2006] BEHNKE, Joachim ; BAUR, Nina ; BEHNKE, Nathalie: *Empirische Methoden der Politikwissenschaft*. Paderborn ; München ua : Schöningh ua, 2006. – ISBN 3506990020

[Benjamins et al. 2011] BENJAMINS, Dr V. R. ; RADOFF, Mark ; DAVIS, Mike ; GREAVES, Mark ; LOCKWOOD, Rose ; CONTRERAS, Dr J.: Semantic Technology Adoption: A Business Perspective. Version: Januar 2011. http://link.springer.com/referenceworkentry/10.1007/978-3-540-92913-0_15. In: DOMINGUE, John (Ed.) ; FENSEL, Dieter (Ed.) ; HENDLER, James A. (Ed.): *Handbook of Semantic Web Technologies*. Springer Berlin Heidelberg, Januar 2011. – ISBN 978–3–540–92912–3, 978–3–540–92913–0, 619–657

[Bergmann 2009] BERGMANN, Michael: *Advantages and Myths of RDF*. http://www.mkbergman.com/483/advantages-and-myths-of-rdf/. Version: 2009

[Berners-Lee 2000] BERNERS-LEE, Tim: *Semantic Web - XML2000*. http://www.w3.org/2000/Talks/1206-xml2k-tbl/Overview.html. Version: 2000

[Berners-Lee 2006] BERNERS-LEE, Tim: Linked Data - Design Issues. In: *w3c.org* (2006). http://www.w3.org/DesignIssues/LinkedData.html

[Berners-Lee and Cailliau 1990] BERNERS-LEE, Tim ; CAILLIAU, Robert: WorldWideWeb: Proposal for a HyperText project / W3C. Version: 1990. http://www.w3.org/Proposal.html. 1990. – Report

[Berners-Lee et al. 2001] BERNERS-LEE, Tim ; HENDLER, James ; LASSILA, Ora: The Semantic Web. In: *The Scientific American* (2001), Nr. May. http://www.scientificamerican.com/article/the-semantic-web/

[Bizer et al. 2013] BIZER, Christian ; ECKERT, Kai ; MEUSEL, Robert ; MÜHLEISEN, Hannes ; SCHUHMACHER, Michael ; VÖLKER, Johanna: Deployment of RDFa, Microdata, and Microformats on the Web textendash A Quantitative Analysis. Version: Januar 2013. http://link.springer.com/chapter/10.1007/978-3-642-41338-4_2. In: ALANI, Harith (Ed.) ; KAGAL, Lalana (Ed.) ; FOKOUE, Achille (Ed.) ; GROTH, Paul (Ed.) ; BIEMANN, Chris (Ed.) ; PARREIRA, Josiane X. (Ed.) ; AROYO, Lora (Ed.) ; NOY, Natasha (Ed.) ; WELTY, Chris (Ed.) ; JANOWICZ, Krzysztof (Ed.): *The Semantic Web textendash ISWC 2013*. Springer Berlin Heidelberg, Januar 2013 (Lecture Notes in Computer Science 8219). – ISBN 978–3–642–41337–7, 978–3–642–41338–4, 17–32

[Bizer et al. 2009] BIZER, Christian ; HEATH, Tom ; BERNERS-LEE, Tim: Linked Data - The Story So Far:. In: *International Journal on Semantic Web and Information Sys-*

tems 5 (2009), Nr. 3, 1–22. http://dx.doi.org/10.4018/jswis.2009081901. – DOI 10.4018/jswis.2009081901. – ISSN 1552–6283, 1552–6291

[Blumauer and Pellegrini 2006] BLUMAUER, Andreas ; PELLEGRINI, Tassilo: Semantic Web und semantische Technologien: Zentrale Begriffe und Unterscheidungen. Version: Januar 2006. http://link.springer.com/chapter/10.1007/3-540-29325-6_2. In: PELLEGRINI, Tassilo (Ed.) ; BLUMAUER, Andreas (Ed.): *Semantic Web*. Springer Berlin Heidelberg, Januar 2006 (X.media.press). – ISBN 978–3–540–29324–8, 978–3–540–29325–5, 9–25

[Bratt 2007] BRATT, Steve: Semantic Web, and Other Technologies to Watch. (2007). http://www.w3.org/2007/Talks/0130-sb-W3CTechSemWeb/#(24)

[Bray 2003] BRAY, Tim: The RDF.net Challenge. In: *ongoing by Tim Bray* (2003), Mai. http://www.tbray.org/ongoing/When/200x/2003/05/21/RDFNet

[Breslin et al. 2010] BRESLIN, John G. ; O'SULLIVAN, David ; PASSANT, Alexandre ; VASILIU, Laurentiu: Semantic Web computing in industry. In: *Computers in Industry* 61 (2010), Oktober, Nr. 8, 729–741. http://dx.doi.org/10.1016/j.compind.2010.05.002. – DOI 10.1016/j.compind.2010.05.002. – ISSN 0166–3615

[Budd 2007] BUDD, Andy: *Heuristics for Modern Web Application Development*. http://www.andybudd.com/archives/2007/01/heuristics_for_modern_web_application_development/. Version: Januar 2007

[Bush 1945] BUSH, Vannevar: As we may think. In: *The Atlantic Monthly* 176 (1945), Nr. July, 101–108. http://www.theatlantic.com/magazine/archive/1945/07/as-we-may-think/303881/

[Cardoso 2007] CARDOSO, Jorge: The semantic web vision: Where are we? In: *Intelligent Systems, IEEE* 22 (2007), Nr. 5, S. 84–88

[Codd 1970] CODD, E. F.: A Relational Model of Data for Large Shared Data Banks. In: *Commun. ACM* 13 (1970), Juni, Nr. 6, 377–387. http://dx.doi.org/10.1145/362384.362685. – DOI 10.1145/362384.362685. – ISSN 0001–0782

[Corlosquet et al. 2009a] CORLOSQUET, Stéphane ; DELBRU, Renaud ; CLARK, Tim ; POLLERES, Axel ; DECKER, Stefan: Produce and Consume Linked Data with Drupal! Version: Januar 2009. http://link.springer.com/chapter/10.1007/

978-3-642-04930-9_48. In: BERNSTEIN, Abraham (Ed.) ; KARGER, David R. (Ed.) ; HEATH, Tom (Ed.) ; FEIGENBAUM, Lee (Ed.) ; MAYNARD, Diana (Ed.) ; MOTTA, Enrico (Ed.) ; THIRUNARAYAN, Krishnaprasad (Ed.): *The Semantic Web - ISWC 2009*. Springer Berlin Heidelberg, Januar 2009 (Lecture Notes in Computer Science 5823). – ISBN 978–3–642–04929–3, 978–3–642–04930–9, 763–778

[Corlosquet et al. 2009b] CORLOSQUET, Stéphane ; POLLERES, Axel ; CYGANIAK, Richard ; DECKER, Stefan: Semantic Web Publishing with Drupal / DERI. Version: 2009. https://www.deri.ie/content/semantic-web-publishing-drupal. 2009. – Report

[Cyganiak et al. 2014] CYGANIAK, Richard ; WOODS, David D. ; LANTHALER, Markus: RDF 1.1 Concepts and Abstract Syntax: W3C Recommendation 25 February 2014 / W3C. Version: Februar 2014. http://www.w3.org/TR/rdf11-concepts/#section-Graph-Literal. 2014. – Report

[d'Aquin et al. 2011] D'AQUIN, Dr M. ; DING, Li ; MOTTA, Enrico: Semantic Web Search Engines. Version: Januar 2011. http://link.springer.com/referenceworkentry/10.1007/978-3-540-92913-0_16. In: DOMINGUE, John (Ed.) ; FENSEL, Dieter (Ed.) ; HENDLER, James A. (Ed.): *Handbook of Semantic Web Technologies*. Springer Berlin Heidelberg, Januar 2011. – ISBN 978–3–540–92912–3, 978–3–540–92913–0, 659–700

[d'Aquin et al. 2007] D'AQUIN, Mathieu ; GRIDINOC, Laurian ; ANGELETOU, Sofia ; SABOU, Marta ; MOTTA, Enrico: Watson: a gateway for next generation semantic web applications. Busan, Korea, 2007

[d'Aquin et al. 2008] D'AQUIN, Mathieu ; SABOU, Marta ; MOTTA, Enrico ; ANGELETOU, Sofia ; GRIDINOC, Laurian ; LOPEZ, Vanessa ; ZABLITH, Fouad: What can be done with the Semantic Web? An overview of Watson-based applications. In: *CEUR Workshop Proceedings* Bd. 426. Rome, Italy, 2008

[Davis et al. 2013] DAVIS, Ian ; STEINER, Thomas ; LE HORS, Arnoud J.: RDF 1.1 JSON Alternate Serialization (RDF/JSON): W3C Working Group Note 07 November 2013 / W3C. Version: November 2013. http://www.w3.org/TR/rdf-json/. 2013. – Report

[Ding et al. 2004] DING, Li ; FININ, Tim ; JOSHI, Anupam ; PAN, Rong ; COST, R. S. ; PENG, Yun ; REDDIVARI, Pavan ; DOSHI, Vishal ; SACHS, Joel: Swoogle: A Search and Metadata Engine for the Semantic Web. In: *Proceedings of the Thirteenth ACM*

International Conference on Information and Knowledge Management. New York, NY, USA : ACM, 2004 (CIKM '04). – ISBN 1–58113–874–1, 652–659

[Dodds 2010] DODDS, Leigh: RDF and JSON: A Clash of Model and Syntax. In: *Lost Boy* (2010), Dezember. http://blog.ldodds.com/2010/12/02/rdf-and-json-a-clash-of-model-and-syntax/

[Domingue et al. 2011] DOMINGUE, John ; FENSEL, Dieter ; HENDLER, James A.: Introduction to the Semantic Web Technologies. Version: Januar 2011. http://link.springer.com/referenceworkentry/10.1007/978-3-540-92913-0_1. In: DOMINGUE, John (Ed.) ; FENSEL, Dieter (Ed.) ; HENDLER, James A. (Ed.): *Handbook of Semantic Web Technologies*. Springer Berlin Heidelberg, Januar 2011. – ISBN 978–3–540–92912–3, 978–3–540–92913–0, 1–41

[Erdmann and Waterfeld 2012] ERDMANN, Michael ; WATERFELD, Walter: Overview of the NeOn Toolkit. Version: Januar 2012. http://link.springer.com/chapter/10.1007/978-3-642-24794-1_13. In: SUÁREZ-FIGUEROA, Mari C. (Ed.) ; GÓMEZ-PÉREZ, Asunción (Ed.) ; MOTTA, Enrico (Ed.) ; GANGEMI, Aldo (Ed.): *Ontology Engineering in a Networked World*. Springer Berlin Heidelberg, Januar 2012. – ISBN 978–3–642–24793–4, 978–3–642–24794–1, 281–301

[Franzon 2014] FRANZON, Eric: *End of Support for the Sindice.com search engine: history, lessons learned, and legacy*. http://semanticweb.com/end-support-sindice-com-search-engine-history-lessons-learned-legacy-guest-post_b42797. Version: Mai 2014

[Gandon and Schreiber 2014] GANDON, Fabien ; SCHREIBER, Guus: RDF 1.1 XML Syntax: W3C Recommendation 25 February 2014 / W3C. Version: 2014. http://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/. 2014. – Report

[Gennari et al. 2003] GENNARI, John H. ; MUSEN, Mark A. ; FERGERSON, Ray W. ; GROSSO, William E. ; CRUBÉZY, Monica ; ERIKSSON, Henrik ; NOY, Natalya F. ; TU, Samson W.: The evolution of Protégé: an environment for knowledge-based systems development. In: *International Journal of Human-Computer Studies* 58 (2003), Januar, Nr. 1, 89–123. http://dx.doi.org/10.1016/S1071-5819(02)00127-1. – DOI 10.1016/S1071–5819(02)00127–1. – ISSN 1071–5819

[Gottron et al. 2012] GOTTRON, Thomas ; SCHERP, Ansgar ; KRAYER, Bastian ; PETERS, Arne: Get the google feeling: Supporting users in findingtextendashrelevant sources

of linked open data at web-scale. In: *Semantic Web Challenge, Submission to the Billion Triple Track* (2012)

[Grimmes et al. 2012] GRIMMES, Gunnar A. ; HARTIG, Olaf ; KIESEL, Malte ; LIWICKI, Marcus: Linked Open Data, Semantic Web Datensätze. Version: Januar 2012. http://link.springer.com/chapter/10.1007/978-3-8274-2664-2_7. In: DENGEL, Andreas (Ed.): *Semantische Technologien*. Spektrum Akademischer Verlag, Januar 2012. – ISBN 978–3–8274–2663–5, 978–3–8274–2664–2, 183–203

[Group 2012] GROUP, W3C OWL W.: OWL 2 Web Ontology Language Document Overview (Second Edition): W3C Recommendation 11 December 2012 / W3C. Version: Dezember 2012. http://www.w3.org/TR/owl2-overview/#ack. 2012. – Report

[Haase et al. 2008] HAASE, Peter ; LEWEN, Holger ; STUDER, Rudi ; TRAN, Duc T. ; ERDMANN, Michael ; D'AQUIN, Mathieu ; MOTTA, Enrico: The neon ontology engineering toolkit. In: *2008 Developers Track*, 2008

[Hartig et al. 2009] HARTIG, Olaf ; BIZER, Christian ; FREYTAG, Johann-Christoph: Executing SPARQL Queries over the Web of Linked Data. Version: Januar 2009. http://link.springer.com/chapter/10.1007/978-3-642-04930-9_19. In: BERNSTEIN, Abraham (Ed.) ; KARGER, David R. (Ed.) ; HEATH, Tom (Ed.) ; FEIGENBAUM, Lee (Ed.) ; MAYNARD, Diana (Ed.) ; MOTTA, Enrico (Ed.) ; THIRUNARAYAN, Krishnaprasad (Ed.): *The Semantic Web - ISWC 2009*. Springer Berlin Heidelberg, Januar 2009 (Lecture Notes in Computer Science 5823). – ISBN 978–3–642–04929–3, 978–3–642–04930–9, 293–309

[Havlik 2011] HAVLIK, Denis: Building Environmental Semantic Web Applications with Drupal. Version: Januar 2011. http://link.springer.com/chapter/10.1007/978-3-642-22285-6_42. In: HREBICEK, Jiri (Ed.) ; SCHIMAK, Gerald (Ed.) ; DENZER, Ralf (Ed.): *Environmental Software Systems. Frameworks of eEnvironment*. Springer Berlin Heidelberg, Januar 2011 (IFIP Advances in Information and Communication Technology 359). – ISBN 978–3–642–22284–9, 978–3–642–22285–6, 385–397

[Heath and Bizer 2011] HEATH, Tom ; BIZER, Christian: Linked Data: Evolving the Web into a Global Data Space. In: *Synthesis Lectures on the Semantic Web: Theory and Technology* 1 (2011), Februar, Nr. 1, 1–136. http://dx.doi.org/10.2200/

S00334ED1V01Y201102WBE001. – DOI 10.2200/S00334ED1V01Y201102WBE001. – ISSN 2160–4711, 2160–472X

[Hendler 2001] HENDLER, James: Agents and the semantic web. In: *IEEE Intelligent systems* 16 (2001), Nr. 2, S. 30–37

[Hickson 2013] HICKSON, Ian: HTML Microdata: W3C Working Group Note 29 October 2013 / W3C. Version: Oktober 2013. http://www.w3.org/TR/2013/NOTE-microdata-20131029/. 2013. – Report

[Holzinger 2005] HOLZINGER, Andreas: Usability Engineering Methods for Software Developers. In: *Commun. ACM* 48 (2005), Januar, Nr. 1, 71–74. http://dx.doi.org/10.1145/1039539.1039541. – DOI 10.1145/1039539.1039541. – ISSN 0001–0782

[Janev and Vranes 2009] JANEV, Valentina ; VRANES, Sanja: Semantic Web Technologies: Ready for Adoption? In: *IT Professional* 11 (2009), Nr. 5, S. 8–16

[Jentzsch et al. 2011] JENTZSCH, Anja ; CYGANIAK, Richard ; BIZER, Chris: State of the LOD Cloud. Version: 2011. http://lod-cloud.net/state/. 2011. – Report

[Johnston and Webber 2005] JOHNSTON, Bill ; WEBBER, Sheila: As we may think: Information literacy as a discipline for the information age. In: *Research Strategies* 20 (2005), Nr. 3, 108–121. http://dx.doi.org/10.1016/j.resstr.2006.06.005. – DOI 10.1016/j.resstr.2006.06.005. – ISSN 0734–3310

[Kaufmann and Bernstein 2007] KAUFMANN, Esther ; BERNSTEIN, Abraham: How Useful Are Natural Language Interfaces to the Semantic Web for Casual End-Users? Version: Januar 2007. http://link.springer.com/chapter/10.1007/978-3-540-76298-0_21. In: ABERER, Karl (Ed.) ; CHOI, Key-Sun (Ed.) ; NOY, Natasha (Ed.) ; ALLEMANG, Dean (Ed.) ; LEE, Kyung-Il (Ed.) ; NIXON, Lyndon (Ed.) ; GOLBECK, Jennifer (Ed.) ; MIKA, Peter (Ed.) ; MAYNARD, Diana (Ed.) ; MIZOGUCHI, Riichiro (Ed.) ; SCHREIBER, Guus (Ed.) ; CUDRÉ-MAUROUX, Philippe (Ed.): *The Semantic Web*. Springer Berlin Heidelberg, Januar 2007 (Lecture Notes in Computer Science 4825). – ISBN 978–3–540–76297–3, 978–3–540–76298–0, 281–294

[Kiesel and Grimnes 2010] KIESEL, Malte ; GRIMNES, Gunnar A.: DBTropestextemdasha linked data wrapper approach incorporating community feedback. In: *Proceedings of EKAW*, Citeseer, 2010

[Kiryakov et al. 2005] KIRYAKOV, Atanas ; OGNYANOV, Damyan ; MANOV, Dimitar: OWLIM textendash A Pragmatic Semantic Repository for OWL. Version: Januar 2005. http://link.springer.com/chapter/10.1007/11581116_19. In: DEAN, Mike (Ed.) ; GUO, Yuanbo (Ed.) ; JUN, Woochun (Ed.) ; KASCHEK, Roland (Ed.) ; KRISHNASWAMY, Shonali (Ed.) ; PAN, Zhengxiang (Ed.) ; SHENG, Quan Z. (Ed.): *Web Information Systems Engineering textendash WISE 2005 Workshops*. Springer Berlin Heidelberg, Januar 2005 (Lecture Notes in Computer Science 3807). – ISBN 978–3–540–30018–2, 978–3–540–32287–0, 182–192

[Knublauch et al. 2004] KNUBLAUCH, Holger ; FERGERSON, Ray W. ; NOY, Natalya F. ; MUSEN, Mark A.: The Protégé OWL Plugin: An Open Development Environment for Semantic Web Applications. Version: Januar 2004. http://link.springer.com/chapter/10.1007/978-3-540-30475-3_17. In: MCILRAITH, Sheila A. (Ed.) ; PLEXOUSAKIS, Dimitris (Ed.) ; HARMELEN, Frank v. (Ed.): *The Semantic Web textendash ISWC 2004*. Springer Berlin Heidelberg, Januar 2004 (Lecture Notes in Computer Science 3298). – ISBN 978–3–540–23798–3, 978–3–540–30475–3, 229–243

[Lagoze et al. 2008] LAGOZE, Carl ; SOMPEL, Herber van d. ; JOHNSTON, Pete ; NELSON, Michael ; SANDERSON, Robert ; WARNER, Simeon: ORE User Guide - Resource Map Implementation in RDF/XML / Open Archives Initiative. Version: 2008. http://www.openarchives.org/ore/1.0/rdfxml. 2008. – Report

[Lassila and Swick 1999] LASSILA, Ora ; SWICK, Ralph R.: Resource Description Framework (RDF) model and syntax specification. (1999). http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/

[Lewis 1982] LEWIS, Clayton: *Using the" thinking-aloud" method in cognitive interface design*. IBM TJ Watson Research Center, 1982

[Lidell and Scott 1940] LIDELL, Henry G. ; SCOTT, Robert: sigmaetamualphanutauiotakappaóvarsigma, sēmantikós. In: *A Greek-English Lexicon* (1940). http://www.perseus.tufts.edu/hopper/text?doc=Perseus:text:1999.04.0057:entry=shmantiko/s

[Lilienthal 2014] LILIENTHAL, Svantje: *Tripel Generator: Entwicklung einer Webanwendung zur erleichterten Erstellung von Tripeln für das Semantic Web*. Berlin, Humboldt-Universität zu Berlin, Master Thesis, Juni 2014

[Magee 2011] MAGEE, Liam: Contemporary dilemmas: tables versus webs. In: *Towards A Semantic Web: Connecting Knowledge In Academic Research* (2011), S. 215

[Manola et al. 2014] MANOLA, Frank ; MILLER, Eric ; MCBRIDE, Brian: RDF 1.1 Primer: W3C Working Group Note 24 June 2014 / W3C. Version: Juni 2014. http://www.w3.org/TR/2014/NOTE-rdf11-primer-20140624/. 2014. – Report

[Masinter et al. 2005] MASINTER, Larry ; BERNERS-LEE, Tim ; FIELDING, Roy T.: Uniform Resource Identifier (URI): Generic Syntax. (2005). http://tools.ietf.org/html/rfc3986

[Mika and Potter 2012] MIKA, Peter ; POTTER, Tim: Metadata Statistics for a Large Web Corpus. In: *LDOW* 937 (2012)

[Molich and Nielsen 1990] MOLICH, Rolf ; NIELSEN, Jakob: Improving a Human-computer Dialogue. In: *Commun. ACM* 33 (1990), März, Nr. 3, 338–348. http://dx.doi.org/10.1145/77481.77486. – DOI 10.1145/77481.77486. – ISSN 0001–0782

[Möller 2010a] MÖLLER, Prof Dr-Ing S.: Motivation und Zielsetzung, Qualität und Gebrauchstauglichkeit. Version: Januar 2010. http://link.springer.com/chapter/10.1007/978-3-642-11548-6_1. In: *Quality Engineering*. Springer Berlin Heidelberg, Januar 2010. – ISBN 978–3–642–11547–9, 978–3–642–11548–6, 1–18

[Möller 2010b] MÖLLER, Prof Dr-Ing S.: Usability Engineering. Version: Januar 2010. http://link.springer.com/chapter/10.1007/978-3-642-11548-6_4. In: *Quality Engineering*. Springer Berlin Heidelberg, Januar 2010. – ISBN 978–3–642–11547–9, 978–3–642–11548–6, 57–74

[Morsey et al. 2012] MORSEY, Mohamed ; LEHMANN, Jens ; AUER, Sören ; STADLER, Claus ; HELLMANN, Sebastian: DBpedia and the live extraction of structured data from Wikipedia. In: *Program: electronic library and information systems* 46 (2012), April, Nr. 2, 157–181. http://dx.doi.org/10.1108/00330331211221828. – DOI 10.1108/00330331211221828. – ISSN 0033–0337

[Murray 1993] MURRAY, Liam: From Memex to Hypertext: Vannevar Bush and the Mind's Machine Edited by James M Nyce and Paul Kahn Academic Press, London, 1991. In: *ReCALL* 5 (1993), Nr. 08, S. 36–37. http://dx.doi.org/10.1017/S0958344000005462. – DOI 10.1017/S0958344000005462

[Musen 1989] MUSEN, Mark A.: An editor for the conceptual models of interactive knowledge-acquisition tools. In: *International Journal of Man-Machine Studies* 31 (1989), Dezember, Nr. 6, 673–698. http://dx.doi.org/10.1016/0020-7373(89)90021-7. – DOI 10.1016/0020–7373(89)90021–7. – ISSN 0020–7373

[Newman 2007] NEWMAN, Andrew: A Relational View of the Semantic Web. In: *O'Reilly XML.com* (2007), März. http://www.xml.com/pub/a/2007/03/14/a-relational-view-of-the-semantic-web.html

[Ngonga Ngomo et al. 2013] NGONGA NGOMO, Axel-Cyrille ; BÜHMANN, Lorenz ; UNGER, Christina ; LEHMANN, Jens ; GERBER, Daniel: Sorry, I Don'T Speak SPARQL: Translating SPARQL Queries into Natural Language. In: *Proceedings of the 22Nd International Conference on World Wide Web*. Republic and Canton of Geneva, Switzerland : International World Wide Web Conferences Steering Committee, 2013 (WWW '13). – ISBN 978–1–4503–2035–1, 977–988

[Nielsen 1994] NIELSEN, Jakob: *Interactive Technologies : Usability Engineering*. Saint Louis, MO, USA : Morgan Kaufmann, 1994 http://site.ebrary.com/lib/alltitles/docDetail.action?docID=10712933. – ISBN 9780080520292

[Nixon et al. 2011] NIXON, Lyndon ; VOLZ, Dr R. ; CIRAVEGNA, Fabio ; STUDER, Rudi: Future Trends. Version: Januar 2011. http://link.springer.com/referenceworkentry/10.1007/978-3-540-92913-0_14. In: DOMINGUE, John (Ed.) ; FENSEL, Dieter (Ed.) ; HENDLER, James A. (Ed.): *Handbook of Semantic Web Technologies*. Springer Berlin Heidelberg, Januar 2011. – ISBN 978–3–540–92912–3, 978–3–540–92913–0, 581–618

[Norton 2013] NORTON, Barry: Usage of Linked Data: Introduction and Application Scenarios. (2013). http://de.slideshare.net/EUCLIDproject/usage-of-linked-data-introduction-and-application-scenarios?ref=http://www.euclid-project.eu/modules/course1

[O'Reilly 2005] O'REILLY, Tim: *What Is Web 2.0 : Design Patterns and Business Models for the Next Generation of Software*. 2005 http://www.oreillynet.com/lpt/a/6228

[Oren et al. 2008] OREN, Eyal ; DELBRU, Renaud ; CATASTA, Michele ; CYGANIAK, Richard ; STENZHORN, Holger ; TUMMARELLO, Giovanni: Sindice.com: a document-oriented lookup index for open linked data. In: *International Journal of Metadata,*

Semantics and Ontologies 3 (2008), Januar, Nr. 1, 37–52. http://dx.doi.org/10.1504/IJMSO.2008.021204. – DOI 10.1504/IJMSO.2008.021204

[Oren 1991] OREN, Tim: From Memex to Hypertext. Version: 1991. http://dl.acm.org/citation.cfm?id=132180.132199. San Diego, CA, USA : Academic Press Professional, Inc., 1991. – ISBN 0–12–523270–5, 319–338

[Pasin 2011] PASIN, Michele: *Survey of Pythonic tools for RDF and Linked Data programming*. http://www.michelepasin.org/blog/2011/02/24/survey-of-pythonic-tools-for-rdf-and-linked-data-programming/. Version: Februar 2011

[Prud'hommeaux and Seaborne 2008] PRUD'HOMMEAUX, Eric ; SEABORNE, Andy: SPARQL Query Language for RDF: W3C Recommendation 15 January 2008 / W3C. Version: Januar 2008. http://www.w3.org/TR/rdf-sparql-query/. 2008. – Report

[Reiter 1978] REITER, Raymond: On Closed World Data Bases. Version: Januar 1978. http://link.springer.com/chapter/10.1007/978-1-4684-3384-5_3. In: GALLAIRE, Hervé (Ed.) ; MINKER, Jack (Ed.): *Logic and Data Bases*. Springer US, Januar 1978. – ISBN 978–1–4684–3386–9, 978–1–4684–3384–5, 55–76

[Rogers 2003] ROGERS, Everett M.: Innovativeness and Adopter Categories. In: *Diffusion of innovations*. 5. New York : Free Press, 2003, S. 267–299

[Rogers et al. 2011] ROGERS, Yvonne ; SHARP, Helen ; PREECE, Jenny: *Interaction Design: Beyond Human-Computer Interaction*. Auflage: 0003. Chichester, West Sussex, U.K : John Wiley & Sons, 2011. – ISBN 9780470665763

[Salo 2013] SALO, Dorothea: Soylent Semantic Web Is People! In: *SWIB 2013*. Hamburg, Germany, November 2013

[Schmachtenberg et al. 2014a] SCHMACHTENBERG, Max ; BIZER, Christian ; PAULHEIM, Heiko: Adoption of the Linked Data Best Practices in Different Topical Domains. In: *The Semantic Web - ISWC 2014*. Trentino, Italy : Springer, 2014

[Schmachtenberg et al. 2014b] SCHMACHTENBERG, Max ; PAULHEIM, Heiko ; BIZER, Christian: Adoption of Linked Data Best Practices in Different Topical Domains: Supplementary Material. Version: 2014. http://data.dws.informatik.uni-mannheim.de/lodcloud/2014/ISWC-RDB/. 2014. – Report

[Seaborne et al. 2008] SEABORNE, Andy ; MANJUNATH, Geetha ; BIZER, Chris ; BRESLIN, John ; DAS, Souripriya ; DAVIS, Ian ; HARRIS, Steve ; IDEHEN, Kingsley ; CORBY, Olivier ; KJERNSMO, Kjetil: SPARQL/Update: A language for updating RDF graphs. In: *W3C Member Submission* 15 (2008). http://www.hpl.hp.com/techreports/2007/HPL-2007-102.pdf

[Shekarpour et al. 2013a] SHEKARPOUR, S. ; HOFFNER, K. ; LEHMANN, J. ; AUER, S.: Keyword Query Expansion on Linked Data Using Linguistic and Semantic Features. In: *2013 IEEE Seventh International Conference on Semantic Computing (ICSC)*, 2013, S. 191–197

[Shekarpour et al. 2013b] SHEKARPOUR, Saeedeh ; AUER, Sören ; NGONGA NGOMO, Axel-Cyrille ; GERBER, Daniel ; HELLMANN, Sebastian ; STADLER, Claus: Generating SPARQL queries using templates. In: *Web Intelligence and Agent Systems* 11 (2013), Januar, Nr. 3, 283–295. http://dx.doi.org/10.3233/WIA-130275. – DOI 10.3233/WIA–130275

[Sletten 2014] SLETTEN, Brian: *Keep On Keeping On*. http://semanticweb.com/keep-on-keepin-on_b41339. Version: Januar 2014

[Smets 1990] SMETS, P.: The combination of evidence in the transferable belief model. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 12 (1990), Mai, Nr. 5, S. 447–458. http://dx.doi.org/10.1109/34.55104. – DOI 10.1109/34.55104. – ISSN 0162–8828

[Sporny et al. 2014] SPORNY, Manu ; LONGLEY, Dave ; KELLOGG, Gregg ; LANTHALER, Markus ; LINDSTRÖM, Niklas: JSON-LD 1.0: A JSON-based Serialization for Linked Data. W3C Recommendation 16 January 2014 / W3C. Version: Januar 2014. http://www.w3.org/TR/2014/REC-json-ld-20140116/. 2014. – Report

[Stuart 2011] STUART, David: *Facilitating access to the web of data : a guide for librarians*. 1. publ. London : London : Facet Publ., 2011

[Studer et al. 1998] STUDER, Rudi ; BENJAMINS, V. R. ; FENSEL, Dieter: Knowledge engineering: principles and methods. In: *Data & knowledge engineering* 25 (1998), Nr. 1, S. 161–197

[Tombros and Sanderson 1998] TOMBROS, Anastasios ; SANDERSON, Mark: Advantages of Query Biased Summaries in Information Retrieval. In: *Proceedings of the 21st An-*

nual International ACM SIGIR Conference on Research and Development in Information Retrieval. New York, NY, USA : ACM, 1998 (SIGIR '98). – ISBN 1–58113–015–5, 2–10

Appendix

A Additional Figures

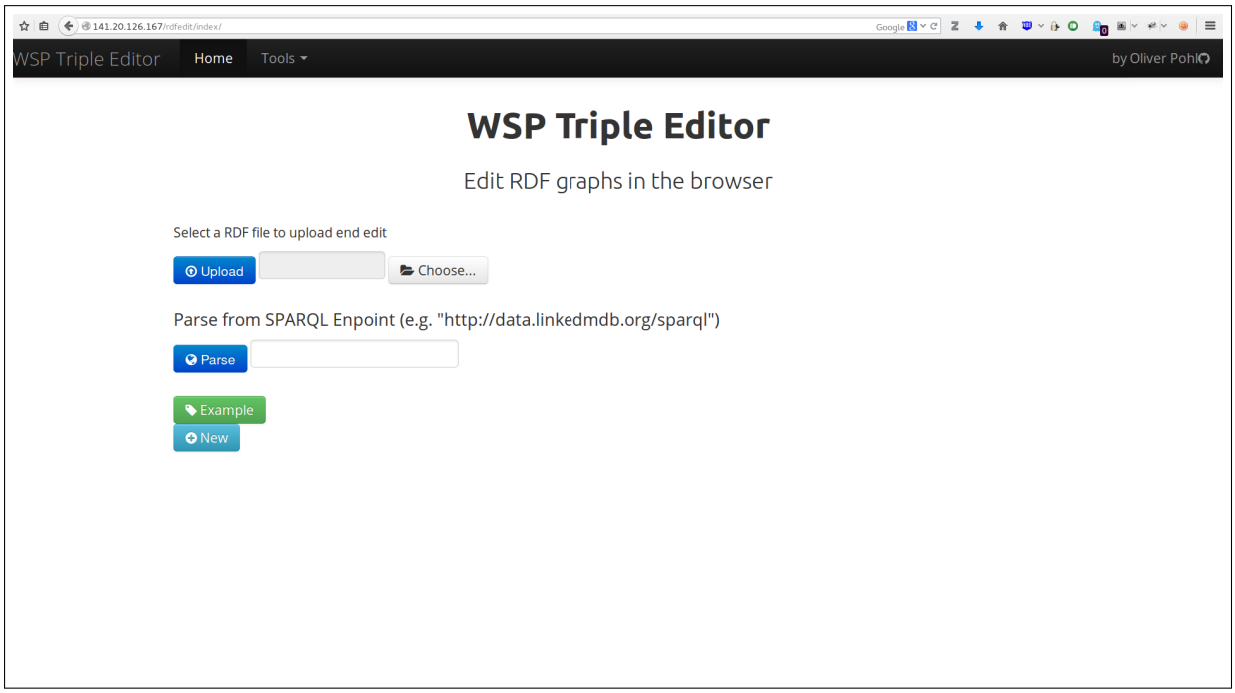

Figure A.1: Start page of rdfedit

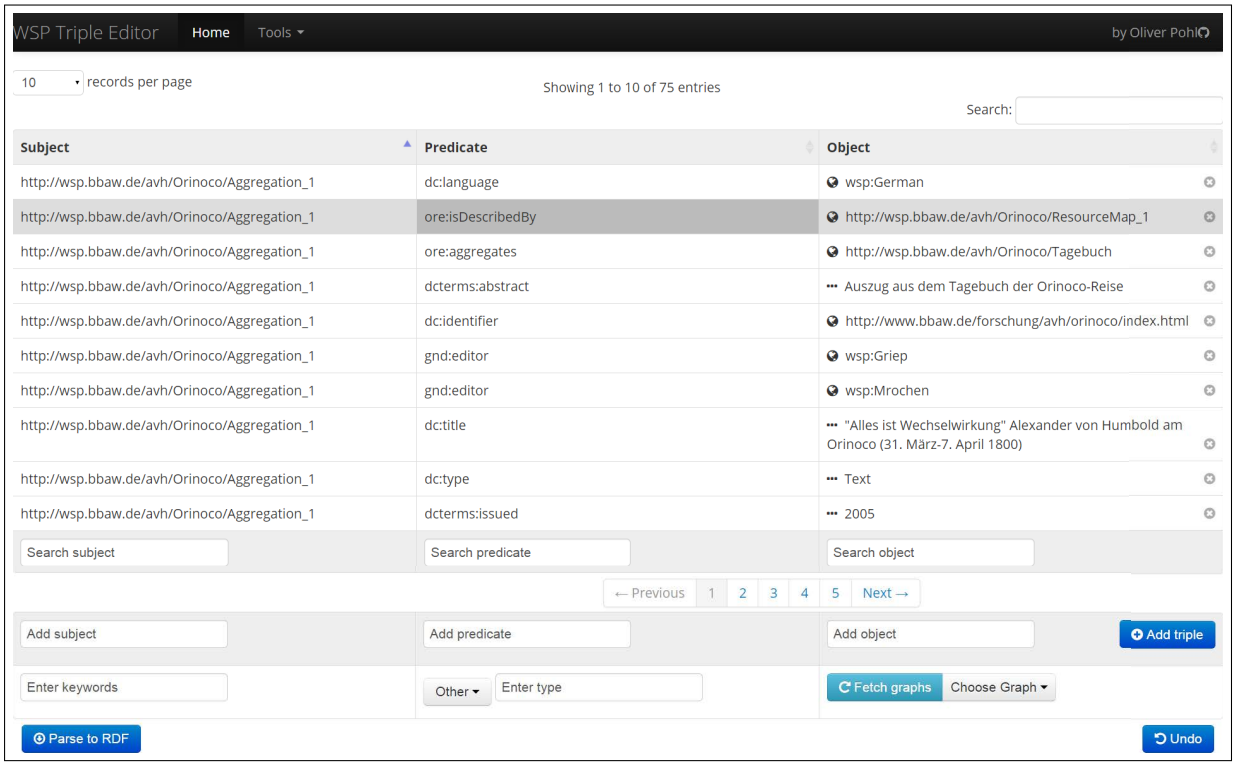

Figure A.2: Tabular interface of rdfedit

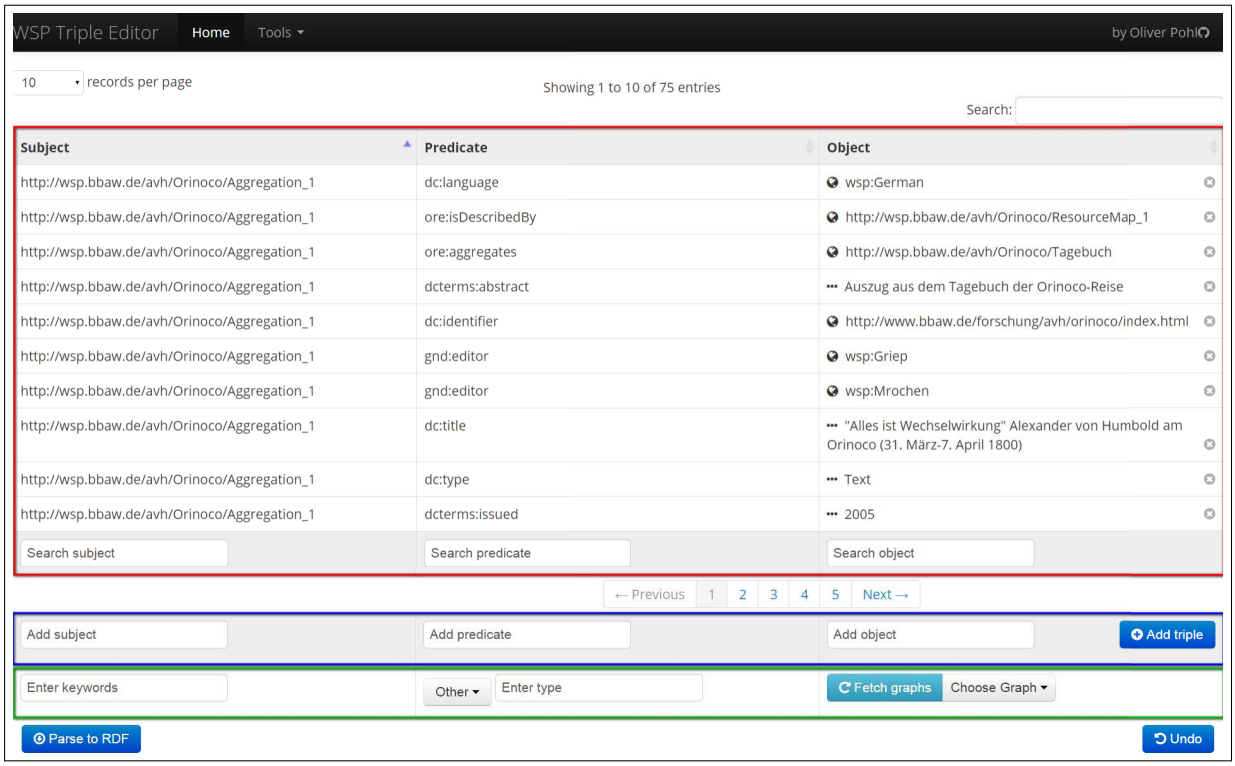

Figure A.3: Tabular interface of rdfedit *with colored markings*

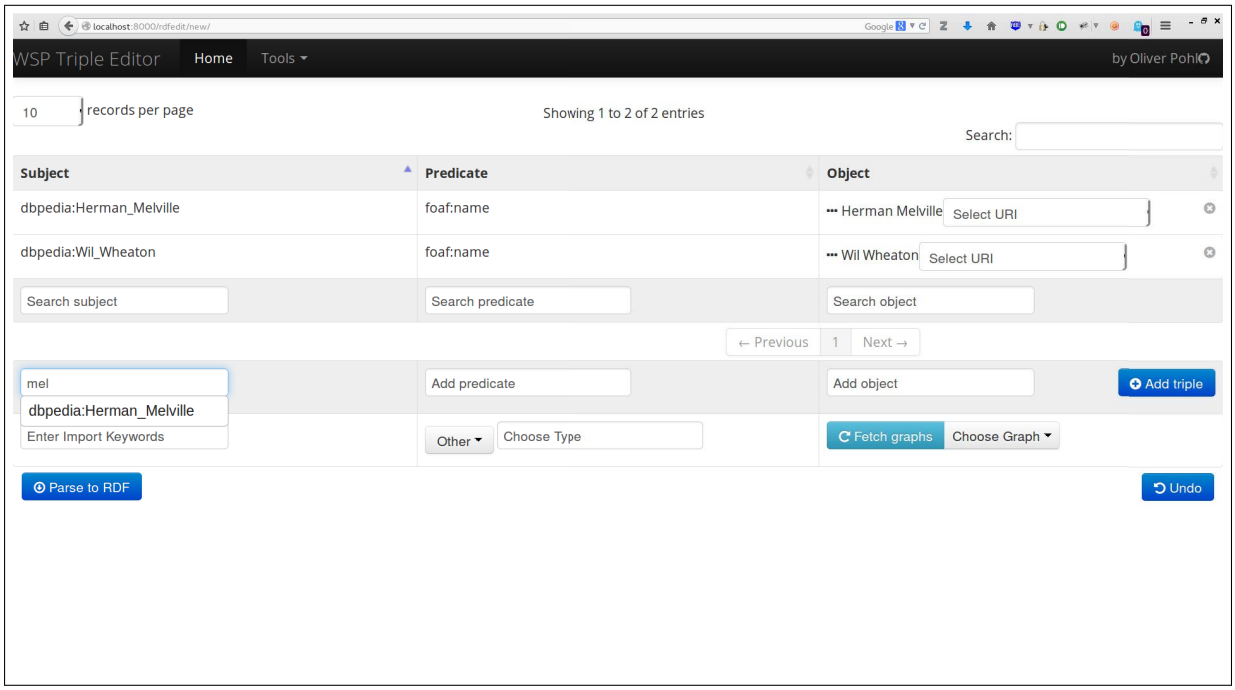

Figure A.4: Auto-completion example when adding a new triple

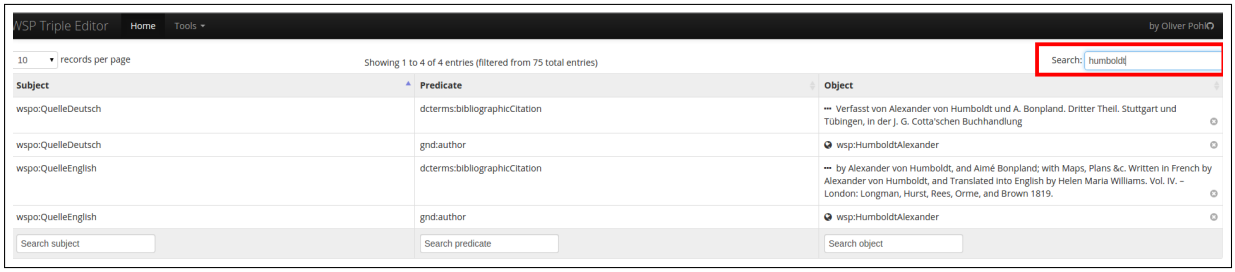

Figure A.5: Extract of the triple_table demonstrating the table-wide search. Only triples containing the string `humboldt` are shown (marked red)

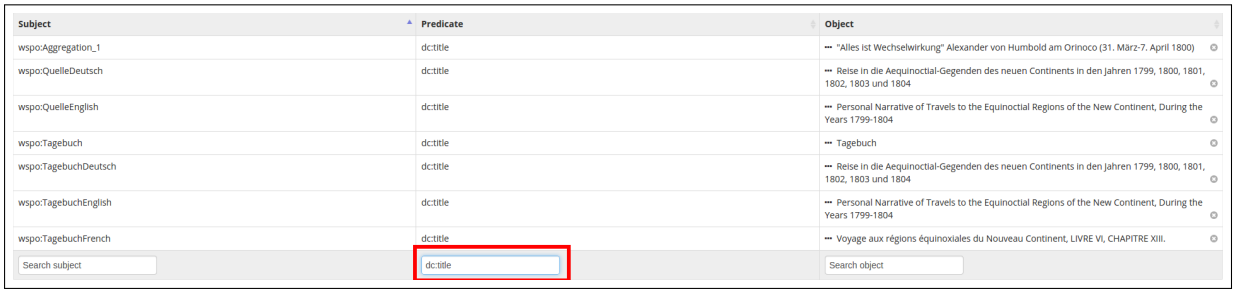

Figure A.6: Extract of the triple_table demonstrating the column search. Only triples where the predicate is (or contains) the string `dc:title` are shown (marked red)

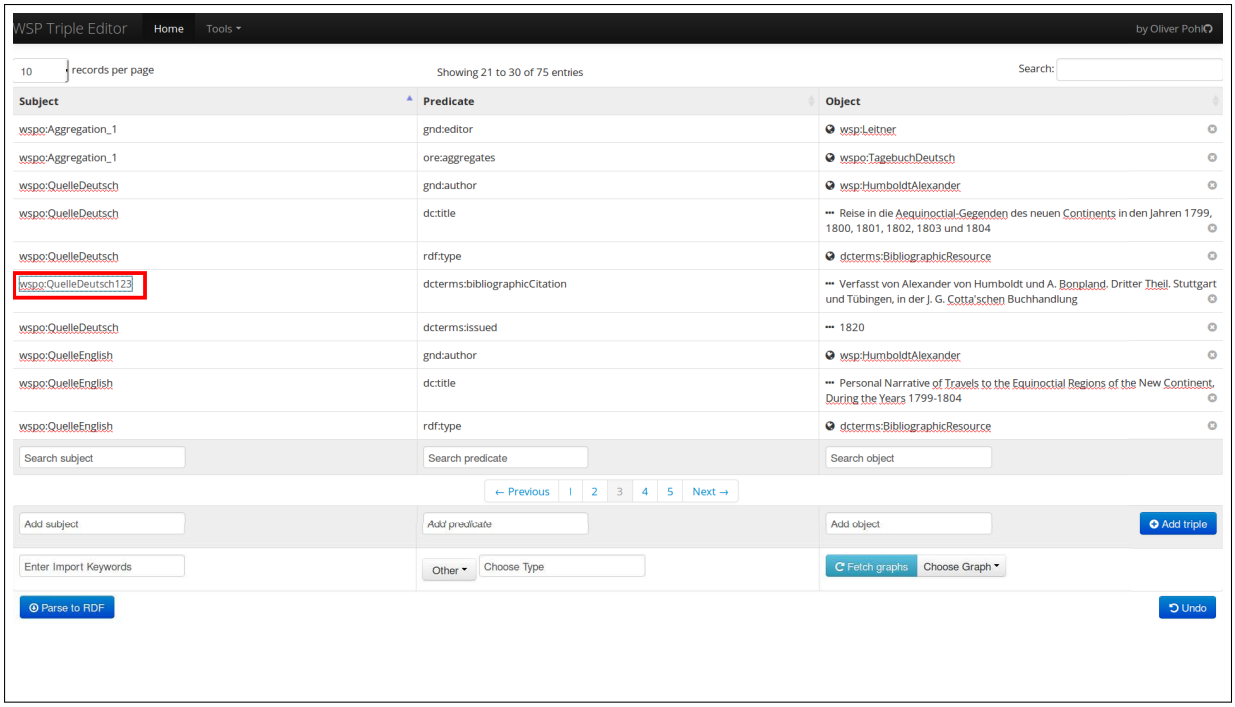

Figure A.7: An edit box appears (marked red) when clicking on a subject or object cell

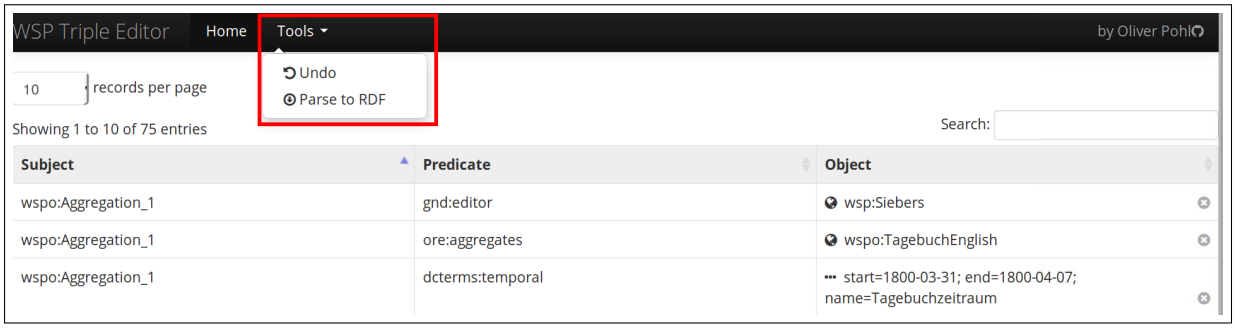

Figure A.8: Tools inside the top-bar of rdfedit*, marked red*

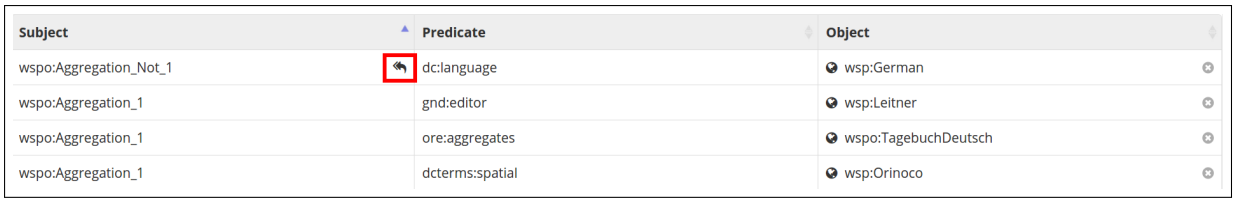

Figure A.9: Excerpt of the triple_table showing the apply-bulk-edit-icon (marked red)

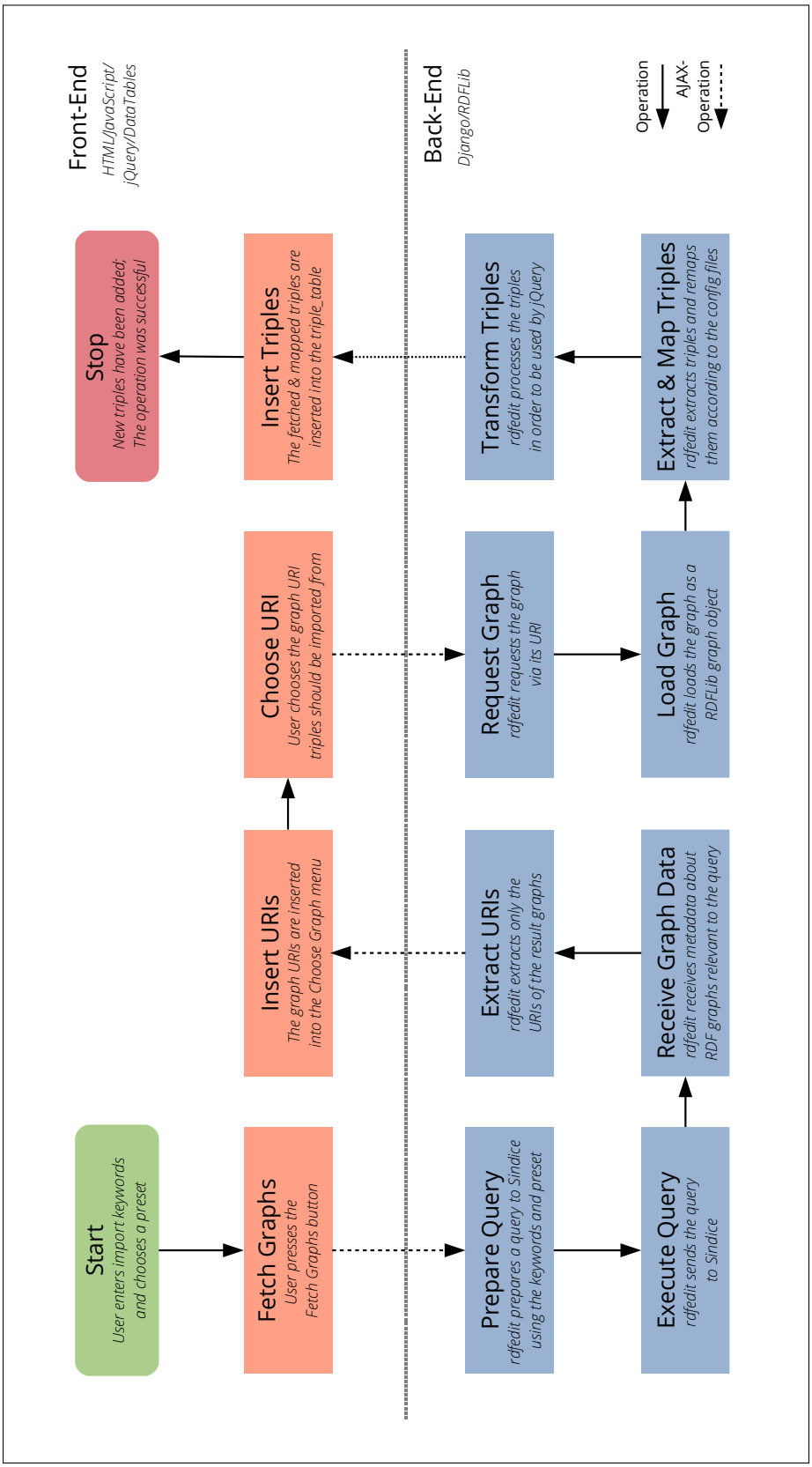

Figure A.10: Flowchart describing the triple import & mapping processes of rdfedit

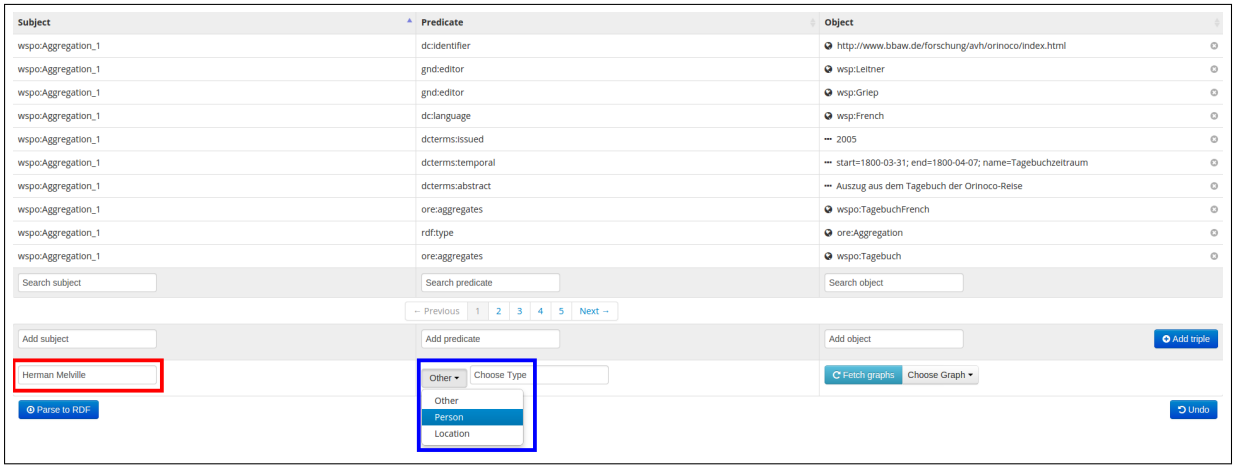

Figure A.11: Initiation of the triple import using the keywords `Herman Melville` (marked red) and the type `Person`

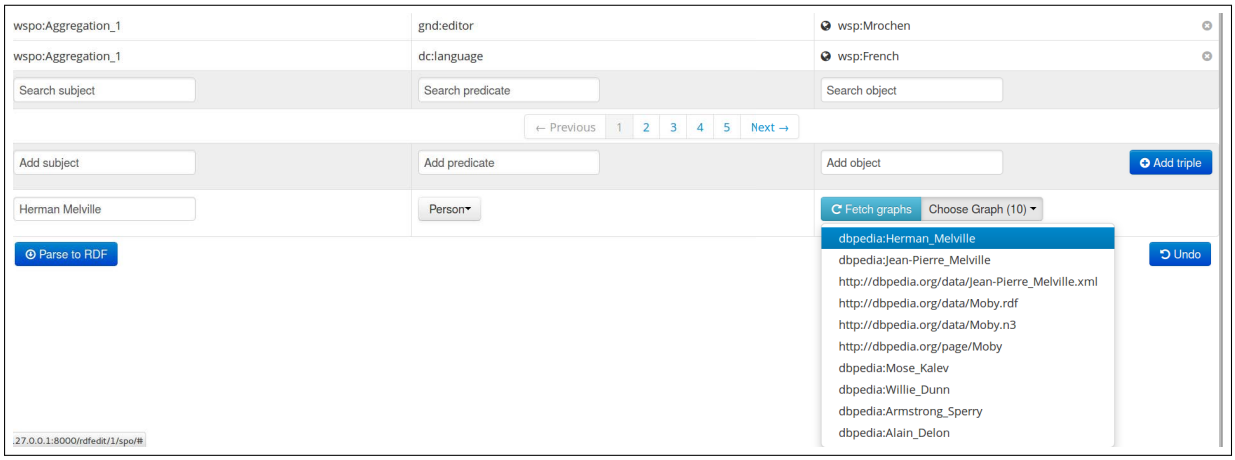

Figure A.12: rdfedit *fetched RDF graph URIs via Sindice. Users can chose among the suggested graphs by pressing the `Choose Graph` button*

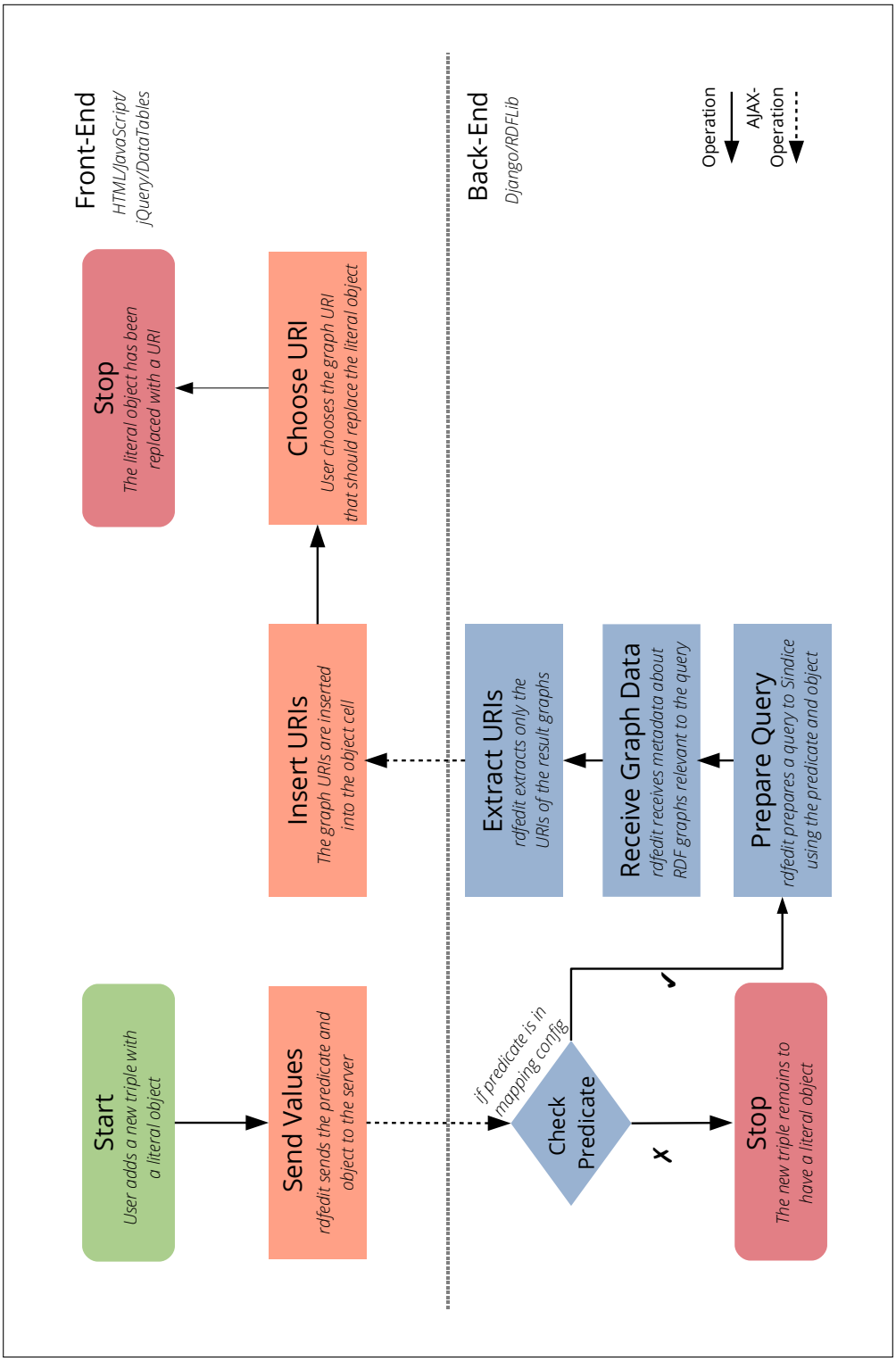

Figure A.13: Flowchart describing the literal-to-URI-conversion function of rdfedit

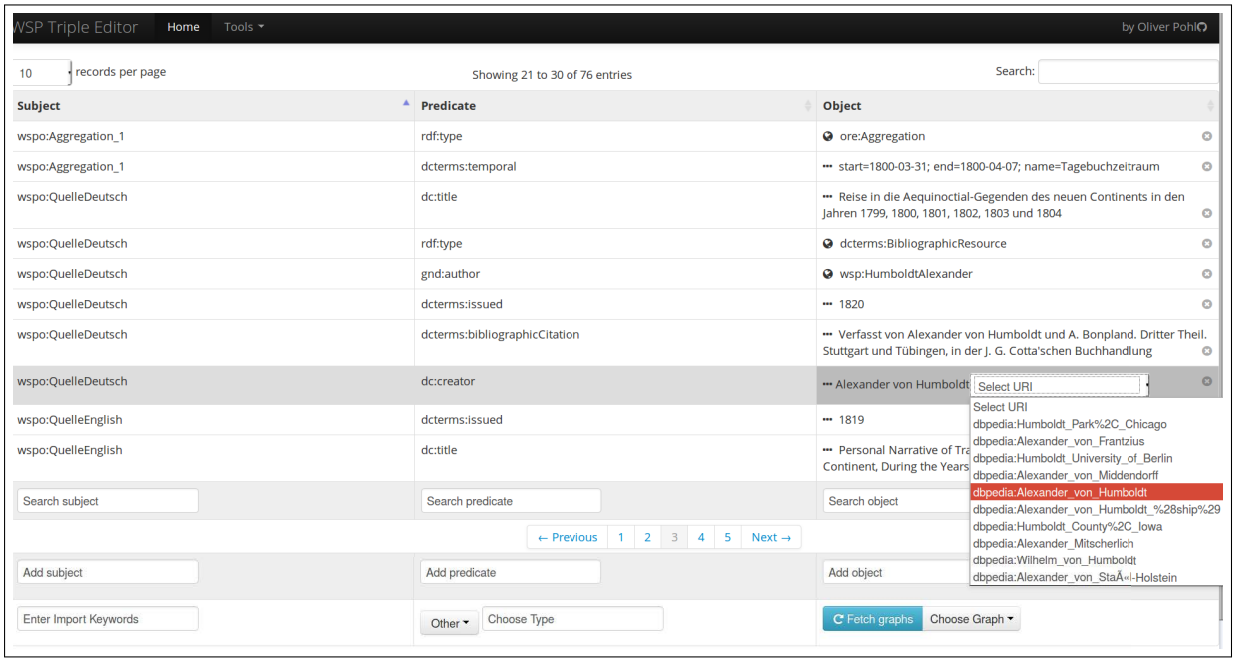

Figure A.14: Screenshot representing the literal-to-URI-conversion function of rdfedit

B Additional Listings

```
NAMESPACES_DICT = {
"dc":"http://purl.org/dc/elements/1.1/",
"DOLCE-Lite":"http://www.loa-cnr.it/ontologies/DOLCE-Lite.owl#",
"foaf":"http://xmlns.com/foaf/0.1/",
"ore":"http://www.openarchives.org/ore/terms/",
"dcmitype":"http://purl.org/dc/dcmitype/",
"rdfs":"http://www.w3.org/2000/01/rdf-schema#",
"xsd":"http://www.w3.org/2001/XMLSchema#",
"owl":"http://www.w3.org/2002/07/owl#",
"rdf":"http://www.w3.org/1999/02/22-rdf-syntax-ns#",
"cidoc_crm_v5":"http://www.cidoc-crm.org/rdfs/cidoc_crm_v5.0.2
   _english_label.rdfs#",
"core":"http://purl.org/vocab/frbr/core#",
"dcterms":"http://purl.org/dc/terms/",
"skos":"http://www.w3.org/2004/02/skos/core#",
"vs":"http://www.w3.org/2003/06/sw-vocab-status/ns#",
"gnd":"http://d-nb.info/standards/elementset/gnd#",
"edm":"http://www.europeana.eu/schemas/edm/",
"wsp":"http://wsp.normdata.rdf/",
"dbpedia": "http://dbpedia.org/resource/",
"dbpprop": "http://dbpedia.org/property/",
"gn": "http://www.geonames.org/ontology#",
"wspo": "http://wsp.bbaw.de/avh/Orinoco/",
"ex": "http://www.example.org/"
}
```

Listing B.1: Namespace dictionary of rdfedit*, accessible in the* `settings.py` *file*

```
function full_to_short_uri(value) {
// Convert full URIs to short URIs
        var short_uri = value;
        $.each(namespaces_dict, function(ns_prefix, ns_full) {
                if (value.match("^"+ns_full)) {
                        short_uri = value.replace(ns_full, ns_prefix + "
                           :");
                }
        });
        return short_uri;
}
```

Listing B.2: JavaScript code to abbreviate long URIs

```
// rdfjson: rdfjson object running in the background of rdfedit
function editSubject(subjectCell) {

        // get old and new subject value, as well as the changed cell
        var oldSubject = "ex:old"; // old value is stored in a global
           variable
        var newSubject = subjectCell.text(); // e.g. "ex:new"

        // get predicate and object value
        var predicate = subjectCell.sibling("#predicate").text();
        var object = subjectCell.sibling("#object").text();


        // create an object container that matches the rdfjson schema
        var objectValues = new Object();
        objectValues["type"] = objectType(object) // literal or URI
        objectValues["value"] = object;

        // Check whether rdfjson already has the new subject
        // If not, add it
        if (!rdfjson.hasOwnProperty(newSubject)) {

                // objects are stored in an array in rdfjson
                var objectArray = new Array(objectValues);

                // create new predicate container
                var predicateObject = new Object();

                // put object array into the predicate container
                predicateObject[predicate] = objectArray;

                // put predicate-object tuple inside the subject node
                rdfjson[newSubject] = predicateObject;

        }

        // Else: newSubject already exists in rdfjson
        else {

                // Check whether the new subject-predicate tuple already
                    exists
                // If not, create and insert it
                if (!rdfjson[newSubject].hasOwnProperty(predicate)) {
                        var objectArray = new Array(objectValues);
                        rdfjson[newSubject][predicate] = objectArray;
                }

                // Else: predicate already exists, so append the object
                   to the objectArray
                else {
                        rdfjson[newSubject][predicate].push(objectValues
                           );
```

```
                }
        }

        // Now delete the original entry (which has been replaced)
        // Check how many objects exist for the oldSubject-predicate-
           tuple
        if (rdfjson[oldSubject][predicate].length == 1) {
                delete rdfjson[oldSubject][predicate];

                // If there aren't any more predicates for oldSubject,
                   delete the oldSubject Node
                if (rdfjson[oldSubject].length == 0) {
                        delete rdfjson[oldSubject]
                }
        }

        // Else if: there are more than 1 objects for the oldSubject-
           predicate tuple
        else if (rdfjson[oldSubject][predicate].length > 1) {

                // create an new object Array
                var objectArray = new Array();

                // iterate over all objects for the oldSubject-predicate
                    tuple and filter the object of the changed triple
                for (var i = 0; i < rdfjson[oldSubject][predicate].
                   length; i++) {
                        var currentObject = rdfjson[oldSubject][
                           predicate][i];
                        if (!currentObject["value"] == object) {
                                objectArray.push(currentObject);
                        }
                }

                // Replace the old objectArray newer and smaller one
                rdfjson[oldSubject][predicate] = objectArray;

        }

}
```

Listing B.3: JavaScript code snippet to apply subject edits to the RDF/JSON object

```
function bulkSubjectEdit(applyIcon) {

        // old subject before its changed is given by a global variable
        var oldSubject = "ex:old";

        // get the (new) subject of the triple the bulk edit should
           originate from
        var newSubject = applyIcon.parent().children("#subject").text()

        // Save all predicate-object tuples for oldSubject
        var rdfjsonSubjectContainer = rdfjson[oldSubject];

        // Replace oldSubject node by newSubject
        rdfjson[newSubject] = rdfjsonSubjectContainer;
        delete rdfedit[oldSubject];

        // Replace all objects that are equal with oldSubject with
           newSubject
        for (subject in rdfjson) {
                for (predicate in rdfjson[subject]) {
                        for (var i = 0; i < rdfjson[subject][predicate].
                           length; i++) {
                                if (rdfjson[subject][predicate][i]["
                                   value"] == oldSubject) {
                                        rdfjson[subject][predicate][i]["
                                           value"] = newSubject;
                                }
                        }
                }
        }

}
```

Listing B.4: JavaScript code snippet to apply a bulk edit to RDF/JSON graph object inside rdfedit

```
function objectEdit(objectCell) {

        // oldObject value is given by a global variable
        var oldObject = "ex:old";

        // get subject, predicate and object of the triple
        var subject = objectCell.sibling("#subject").text();
        var predicate = objectCell.sibling("#predicate").text();
        var object = objectCell.text();

        // Iterate over all objects for the subject-predicate-tuple and
           apply the change
        for (var i = 0; i < rdfjson[subject][predicate].length; i++) {
                if (rdfjson[subject][predicate][i]["value"] == oldObject
                   ) {
                        rdfjson[subject][predicate][i]["value"] =
                           newObject;

                        // Determine type (URI or literal) of the
                           newObject
                        rdfjson[subject][predicate][i]["type"] =
                           objectType(newObject);

                }
        }

}
```

Listing B.5: JavaScript code snippet to apply an object edit to the RDF/JSON graph object inside rdfedit

```
function createObjectContainer(newObject) {

        var objectContainer = new HtmlNode();
        objectContainer.attr("id", "object");

        objectContainer.text(newObject);
        objectContainer.add(deleteIcon);

        return objectContainer;

}

function changeObject(objectCell) {
        // rdfjson stuff ...

        objectCell.insert(createObjectContainer(newObject));

        // ...
}

function addTriple(newSubject, newPredicate, newObject) {

        /// rdfjson stuf ...

        // create new containers
        var subjectContainer = createSubjectContainer(newSubject);
        var predicateContainer = createPredicateContainer(newPredicate);
        var objectContainer = createObjectContainer(newObject);

        // Add new row
        tripleTable.addRow(subjectContainer, predicateContainer,
            objectContainer);

}
```

Listing B.6: Pseudo code snippet to update a cell correctly in rdfedit *by creating a new HTML container*

```
function addTriple(newSubject, newPredicate, newObject) {

        // rdfjson is a global variable

        // function to create an objectContainer (JSON)
        function createObjectContainerJson(newObject) {
                var objectContainer = new Object();
                objectContainer["value"] = newObject;
                objectContainer["type"] = objectType(newObject); //
                    literal or URI

                return objectContainer;
        }

        // check whether newSubject exists in rdfjson
        // if not, create a new node and insert the predicate-object
            tuple
        if (!rdfjson.hasOwnProperty(newSubject)) {

                // create object container
                var objectContainer = createObjectContainerJson(
                    newObject);

                // create a predicate-object-tuple-array and insert the
                    object
                var objectArray = new Array(objectContainer);

                // create predicateContainer for newPredicate and add
                    objectArray as a value
                var predicateContainer = new Object();
                predicateContainer[newPredicate] = objectArray;

                // insert the objectContainer to rdfjson[newSubject][
                    newPredicate]
                rdfjson[newSubject] = predicateContainer;

        }

        // else: newSubject exists
        else {

                // Check if there is already a subject-predicate-tuple
                // if not, add it
                if (!rdfjson[newSubject].hasOwnProperty(predicate)) {

                        // create object Container and Array
                        var objectContainer = createObjectContainerJson(
                            newObject);
                        var objectArray = new Array(objectContainer)

                        // add triple to rdfjon
                        rdfjson[newSubject][newPredicate] = objectArray;
```

```
                }

                // else: newSubject-newPredicate-tuple exists
                else {

                        // create Object container and push it to the
                           newSubject-newPredicate-tuple-array
                        var objectContainer = createObjectContainerJson(
                           newObject);
                        rdfjson[newSubject][newPredicate].push(
                           objectContainer);

                }
        }

}
```

Listing B.7: JavaScript code snippet to add a new triple to the RDF/JSON graph object inside rdfedit

```
function deleteTriple(deleteIcon) {

        var row = deleteIcon.parent();

        // Clicking on the delete icon (in the row) deletes the triple
           in that row
        var subject = row.children("#subject").text();
        var predicate = row.children("#predicate").text();
        var object = row.children("#object").text();

        // rdfjson is a global variable and already given

        // check how many objects exists for the subject-predicate-tuple
        // if there are more than one, simply delete the object
        if (rdfjson[subject][predicate].length > 1) {

                // create a copy of the object array - iterating and
                   deleting directly messes up the loop and is bad
                   programming logic
                var objectArray = rdfjson[subject][predicate];

                // iterate over the array of object-Objects {}
                for (var i=0; i < objectArray.length; i++) {

                        currentObject = objectArray[i];

                        // this: value of the current object {}
                        if (currentObject["value"] == object) {
                                // Remove the object value if it belongs
                                    to the triple that should be deleted
                                rdfjson[subject][predicate].splice(i, 1)
                                   ;
                        }
                });
        }

        // else if there is only one predicate-object-tuple, delete it
        else if (rdfjson[subject][predicate].length == 1) {
                delete rdfjson[subject][predicate];
        }

        // If the subject doesn't have any more predicate-object tuples,
            delete the subject completely
        if (Object.keys(rdfjson[subject]).length == 0){
                delete rdfjson[subject];
        }

}
```

Listing B.8: JavaScript code snippet to delete a triple inside the RDF/JSON graph object inside rdfedit

```
function deleteTriple(deleteIcon) {

        // When clicking on the deleteIcon, the deletion is initiated
        var row = deleteIcon.parent();

        // rdfjson delete operation ...

        // When clicking on
        var rowNum = tripleTable.getRowNumber(row);

        // delete the row
        tripleTable.deleteRow(rowNum);

}
```

Listing B.9: Simplified JavaScript code that deletes a triple in the triple-table

```
"""
This␣python␣function␣builds␣a␣query␣for␣Sindice
"""

# import important constants from rdfedits configuration files
from WSP.settings import SINDICE_CONFIG_QUERY, SINDICE_API_URL

def build_sindice_query(keywords, type):
    # Build the basic query
    query = SINDICE_API_URL

    # Create a query dictionary
    query_dict = dict()

    # Replace the whitespace with +
    query_dict["q"] = keywords.rstrip().replace("␣", "+")
    query_dict["format"] = "json"

    # Read the query config
    query_config = simplejson.loads(open(SINDICE_CONFIG_QUERY, 'r').read
        ())

    if type in query_config:
        type_config = query_config[type]

        for parameter in type_config:
            query_dict[parameter] = type_config[parameter]

    # Iterable variable
    query_parameters_counter = 0

    # Build the query URL by iterating over the query dict
    for query_parameter in query_dict:
```

```
        # increment the iterable
        query_parameters_counter += 1

        # Adapt the query
        if "=" in query_parameter:
            query += query_parameter + ":" + query_dict[query_parameter]
        else:
            query += query_parameter + "=" + query_dict[query_parameter]

        # Add & if appropriate
        if query_parameters_counter < len(query_dict):
            query += "&"

    return query
```

Listing B.10: Python code to build a query for Sindice using the keywords and preset entered by the user


```
{
  "totalResults": 62,
  "author": "Sindice.com",
  "title": "Sindice search: Herman Melville domain:dbpedia class:foaf:
     person",
  "itemsPerPage": 10,
  "startIndex": 0,
  "updated": "2014\/11\/08\/ 12:29:16 +0000",
  "search": "http:\/\/www.sindice.com\/opensearch.xml",
  "base": "http:\/\/api.sindice.com\/v3\/search?q=Herman+Melville&fq=
     domain%3Adbpedia+class%3Afoaf%3Aperson&format=json",
  "entries": [
    {
      "link": "http:\/\/dbpedia.org\/resource\/Herman_Melville",
      "updated": "2010\/06\/24",
      "formats": [
        "RDF"
      ],
      "title": [
        {
          "type": "literal",
          "value": "\u041c\u0435\u043b\u0432\u0438\u043b\u043b, \u0413\
             u0435\u0440\u043c\u0430\u043d"
        },
        {
          "type": "literal",
          "value": "\u8d6b\u5c14\u66fc\u00b7\u6885\u5c14\u7ef4\u5c14"
        },
        {
          "type": "literal",
          "value": "\u30cf\u30fc\u30de\u30f3\u30fb\u30e1\u30eb\u30f4\
             u30a3\u30eb"
        },
        {
```

```
          "type": "literal",
          "value": "Herman Melville"
        }
      ],
      "rank": 1,
    }
  ],
  "query": {
    "startIndex": 0,
    "role": "request",
    "searchTerms": "Herman Melville domain:dbpedia class:foaf:person",
    "responseTime": 4741
  }
}
```


Listing B.11: Example JSON data that is returned from Sindice when looking up relevant RDF graphs for `Herman Melville` of the `Person`

C Statement of Agreement

Sehr geehrte Teilnehmerin, sehr geehrter Teilnehmer,

vielen Dank für Ihre Bereitschaft an diesem Test. Der folgende Versuch findet im Rahmen einer Masterarbeit statt.

Mit dem folgenden Versuch soll die neu entwickelte Webanwendung *rdfedit* hinsichtlich ihrer Gebrauchstauglichkeit (Usability) getestet werden. Zu diesem Zweck wird Ihnen einen Laptop gestellt, auf welcher die Anwendung installiert ist, sodass Sie mit diesem Gerät die vorgegebenen Aufgaben ausführen können. Es steht Ihnen frei, ob Sie ein Touchpad oder eine kabellose Maus zur Bedienung von *rdfedit* verwenden möchten.

rdfedit ist eine Webanwendung, die es Personen mit geringen Semantic-Web-Vorkenntnissen ermöglichen soll, Daten für das Semantic Web zu erzeugen. Das Semantic Web beruht auf der Idee, die Bedeutungen von Aussagen und Informationen für Computer verständlich zu machen und diese Aussagen so automatisch weiterverarbeiten zu können.

Bei diesem Versuch handelt es sich um einen Thinking Aloud Test. Im Laufe des Versuchs werden Ihnen verschiedene Aufgabenstellungen gegeben, welche Sie mit der *rdfedit* Webanwendung bearbeiten sollen. Bitte äußern Sie dem Versuchsleiter Ihre Gedanken, Gefühle und Motive während Sie die vorgegebenen Aufgaben bearbeiten. Für diesen Versuch benötigen Sie keine Vorkenntnisse aus dem Semantic-Web-Bereich. Ihre Aktionen am Computer sowie Ihre Äußerungen werden von einer Software mitgeschnitten sowie vom Versuchsleiter stichpunktartig notiert. Die Aufzeichnungen dienen ausschließlich dem Zweck nach dem Versuch ein Protokoll zu verfassen sowie Ihre Anmerkungen noch einmal genauer hinsichtlich der Usability von *rdfedit* untersuchen zu können. Bitte denken Sie daran, dass bei diesem Versuch die Webanwendung getestet wird, und nicht Sie. Es gibt keine richtigen oder falschen Lösungsmöglichkeiten.

Beispielaufgabe: *Erzeugen Sie ein neues Textdokument.*

Beispieläußerung: „Ich bewege den Mauscursor auf den ‘Neues Dokument’-Button und klicke diesen. Dieser Button ist leicht zu finden.“

Falls Sie zwischendurch Fragen oder Kommentare haben, teilen Sie diese dem Versuchsleiter mit. Dieser wird Ihre Fragen notieren und beantworten. Bitte beachten Sie, dass Ihnen der Versuchsleiter keine Fragen zur Bedienung von *rdfedit* beantworten darf.

Alle erfassten Daten werden selbstverständlich pseudonymisiert und ausschließlich für wissenschaftliche Zwecke verwendet.

Einverständniserklärung

- □ Ich bin über den Sinn und die Durchführung dieses Versuchs im Vornherein informiert worden und gebe hiermit mein Einverständnis zur elektronischen Speicherung und pseudonymisierten Weiterverarbeitung meiner Daten. Die im Versuch erhobenen Daten werden nicht an Dritte weiter gegeben und ausschließlich für wissenschaftliche Zwecke verwendet.
- □ (Optional): Ich gebe darüber hinaus mein Einverständnis, dass anonymsierte Kopien vom audio-visuellen Material, welches bei meiner Teilnahme an diesem Versuch erzeugt wird, an die Gutachter dieser Masterarbeit sowie der an der Humboldt-Universität zu Berlin zuständigen Stelle für die Archivierung von Abschlussarbeiten in Form einer DVD ausgehändigt werden dürfen.

Ort, Datum	Unterschrift

D Experiment Instructions

Sidenote: Since this test was conducted in Germany with German participants, the following test instructions were also handed out in German.

Hintergrund

Das Semantic Web beruht auf der Idee, die Bedeutungen von Aussagen und Informationen für Computer verständlich zu machen und diese Aussagen so automatisch weiterverarbeiten zu können. Diese Aussagen werden in Form von sogenannten Tripeln ausgedrückt, welche aus drei Bestandteilen bestehen: Subjekt (S), Prädikat (P) und Objekt (O). Zusammen ergeben mehrere Tripel einen RDF-Graphen, welcher die gesammelten Aussagen für Mensch und Maschine verständlich zusammenfasst. Objekte können zwei verschiedene Formen annehmen: URIs (Uniform Resource Identifier, ungefähr vergleichbar mit URLs) und Literale. URIs verweisen auf fest definierte Konzepte, die bereits im Semantic Web existieren bzw. mit einem Graphen beschrieben werden sollen (z.B. `ex:Moby_Dick`). Literale sind einfache Werte (z.B. “Tagebuch” oder “1980”), die keine URI sind.

Es folgen nun Beispiele für Tripel mit einem URI- und einem Literal-Objekt:

- **URI-Objekt:** `ex:Moby_Dick dc:type ex:book` — Das Konzept Moby Dick (`ex:Moby_Dick`, = S) ist vom Typ (`dc:type`, = P) Buch `ex:book` (= O).
- **Literal-Objekt:** `ex:Moby_Dick dc:title ‘‘Moby Dick’’` — Das Konzept Moby Dick (`ex:Moby_Dick`, = S) hat den Titel (`dc:title`, = P) `‘‘Moby Dick’’` (= O).

Anhand dieser zwei Tripel kann abgeleitet werden, dass es sich bei dem Konzept `ex:Moby_Dick` um ein Buch handelt und dieses Buch den Titel “Moby Dick” besitzt. Da das `ex:book` auch ein Konzept ist, hat dieses Moby Dick Buch auch alle Eigenschaften eines Buches (Autor, Titel, Seitenanzahl, Verlag usw.).

Die folgenden Seiten enthalten Aufgaben, welche Sie mit Hilfe von rdfedit *lösen sollen. Bitte äußern Sie dem Versuchsleiter Ihre Gedanken, Gefühle und Motive während Sie die vorgegebenen Aufgaben bearbeiten. Bitte äußern Sie sich auch, wenn Sie eine Aufgabe oder Teilaufgabe abgeschlossen haben.*

Anweisung & Aufgaben

1) Sie beginnen auf der Startseite von *rdfedit*. Bitte laden Sie die Datei `Graph.rdf` aus dem Downloads-Ordner von der Festplatte hoch, um diese mit *rdfedit* zu bearbeiten.

2) Sie befinden sich nun in der tabellarischen Editieransicht von *rdfedit*. In dieser Ansicht können Sie sogenannte RDF-Graphen manipulieren und dem Graphen z.B. sogenannte Tripel hinzufügen. Tripel bestehen aus drei Bestandteilen: Subjekt (S), Prädikat (P) und Objekt (O). Bitte fügen Sie die folgenden Tripel hinzu:

    1) `wspo:Aggregation_1 ore:aggregates wspo:TagebuchDeutsch .`
    2) `wspo:QuelleDeutsch dc:language wsp:German .`
    3) `wspo:Tagebuch dc:title ‘‘Tagebuch’’.`

3) Löschen Sie nun die folgenden Tripel:

    1) `wspo:Aggregation_1 gnd:editor wsp:Siebers .`
    2) `wspo:Tagebuch dc:title ‘‘Tagebuch’’.`
    3) Ein Tripel Ihrer Wahl mit `dc:language` als Prädikat.

4) Bitte machen Sie die drei Löschungen wieder rückgängig. Schildern Sie bitte Ihre Vorgehensweise.

5) Ändern Sie den Wert der Objekte der Tripel mit den folgenden Objektwerten. Hängen Sie einfach eine `1` an die jeweiligen Werte:

    1) `wsp:Leitner`
    2) `‘‘Digitale Edition’’`
    3) `‘‘Text’’`

6) Ändern Sie nun die Subjektwerte eines Tripels, welche `wspo:Aggregation_1` als Subjekt besitzen zu `wspo:Aggregation_2`.

7) Bitte übernehmen Sie die im vorangegangenen Punkt genannte Änderung für alle Tripel mit dem selben Subjektwert. Schildern Sie bitte Ihre Vorgehensweise.

8) *rdfedit* bietet die Möglichkeit, bereits existierende Tripel aus externen Quellen zu importieren. Bei diesem Import-Prozess wird nach bereits existierenden RDF-Graphen gesucht, welche zu bestimmten Stichworten relevante Tripel enthalten. Bitte starten Sie einen solchen Import-Prozess mit den folgenden Stichworten und Typen. Wählen Sie anschließend den Graphen aus, welcher Ihnen am geeignetsten zum Tripel-Import erscheint. Bitte vergessen Sie nicht, Ihre Vorgehensweise zu schildern.

    1) Stichworte: Wil Wheaton — Typ: Person
    2) Stichworte: Herman Melville — Typ: Person
    3) Stichwort: Berlin — Typ: Location

9) Wenn neue Tripel manuell hinzugefügt werden, bietet *rdfedit* unter bestimmten Umständen an, literale Objekte (z.B. `Wil Wheaton`) durch passende URIs (z.B. `dbpeadia:Wil_Wheaton`) zu ersetzen. Bitte fügen Sie nun die folgenden Tripel hinzu und ersetzen Sie die literalen Objekte durch eine möglichst geeignete URI (bzw. URL).

    1) `wspo:Aggregation_1 dcterms:spatial ‘‘Berlin’’.`
    2) `wspo:TagebuchDeutsch dc:creator ‘‘Alexander von Humboldt’’.`
    3) `wspo:TagebuchEnglish dc:creator ‘‘Wil Wheaton’’.`

10) Speichern Sie den Graphen samt aller vollzogenen Änderungen auf der Festplatte im Downloads-Ordner.

11) Sie haben nun alle Aufgaben bearbeitet. Nun haben Sie die Möglichkeit, weitere Anmerkungen zu *rdfedit* zu machen. Was hat Ihnen gefallen bzw. nicht gefallen. Fiel es Ihnen leicht die Bedienung von *rdfedit* zu erlernen?

Vielen Dank für Ihre Teilnahme an diesem Versuch.

E Thinking Aloud Test Transcripts

Participant A (Expert I)

1) **Upload**
   - Successful file choice and upload
   - Remark: Choose and upload are not intertwined good enough.
2) **Add**
   - Add environment is not instantly detected
   - Triple addition is successful, auto-completion is used
   - Remark: Auto-complete is a good feature.
   - Bug: Place holder values are not always deleted
3) **Delete**
   - Multiple column-searches are being used
   - Delete icon is detected instantly
   - Successful deletion of triples
4) **Undo**
   - Option to undo operations is searched but not found instantly
   - Found undo option in top bar after a few seconds
   - Bug: Top bar undo option only reverts one operation
   - Question: Is there a keyboard shortcut like `Ctrl+Z` to undo changes?
5) **Object Edit**
   - Tries to click into area around the value
   - Remark: Double clicking does not activate edit mode.
   - Remark: Enter does not apply changes.
   - Edit is successfully applied after a few tries

6) **Subject Edit**

- Edit is successfully applied instantly

7) **Bulk Edit**

- Question: Should every single triple with that value be deleted manually?
- Apply-to-all icon is being interpreted as an undo icon
- Still presses the icon anyway
- Bug: One subject value remains unchanged
- Remark: Bulk edit is helpful, but the feature is not presented apparent enough.

8) **Triple Import**

- Enter import keywords field is detected instantly
- Values are being entered correctly
- Chooses the correct labels using the Choose Type button
- Fetches Graphs
- Error: DBpedia not reachable
- Question: Is there a cancel operation button?

9) **Literal-to-URI-conversion**

- Successfully adds new triples with literal objects
- Recognizes the select URI dropdown list in the new object cells
- Remark: That feature is very helpful and well understandable.

10) **Export/Download**

- Options to download the files not instantly detected
- Parse to RDF button detected on the bottom of *rdfedit* a few seconds later

11) **Other Remarks**

- Good: easy to operate, stable, search, sort, import features
- Suggestions: Label actions better, rename “Parse to RDF”
- Missing: Expert view

Participant B (Novice I)

1) **Upload**
    - Chooses file from hard drive and uploads it
2) **Add**
    - Remark: Interface is now being displayed.
    - Add environment is instantly detected
    - Auto-completion is being used
    - Question: Where do the suggestions from the auto-completion come from?
    - Suggestion: Sort the auto-completion list items alphabetically and by length
3) **Delete**
    - Uses sort function
    - Uses the table-wide search to filter triples
    - Instantly recognizes delete icon and deletes the triples
    - Suggestion: Display another confirmation whether triple really should be deleted
    - Question: Is there an option to only search for literals?
4) **Undo**
    - Finds undo option inside top bar and uses it
    - Question: Is there a recycling bin?
5) **Object Edit**
    - Uses table-wide and column searches to filter triples
    - Remark: Column searches are very convenient.
    - Tries to double click the objects but that doesn't work
    - In the end figures out how to edit the objects
6) **Subject Edit**
    - Has no problems and applies the method learned previously

7) **Bulk Edit**

- Bulk edit icon is being recognized but understood as an undo icon
- Had to abort the task and be shown what the icon does
- Remark: Bulk edit is a helpful feature but the icon used was not a good choice.

8) **Triple Import**

- Import environment is instantly detected
- Enters keywords and uses the Choose Graphs menu
- Imports graphs
- Question: What are the sources for that import?
- Remark: Feature is nice.
- Suggestion: Show highlights longer and display imports separately

9) **Literal-to-URI-Conversion**

- Adds new triples without problems
- Chooses appropriate URIs to replace literals with
- Remark: Other URIs suggested by the conversion feature than by the triple import feature.
- Remark: The table does not focus on the converted object anymore after the conversion has taken place.
- Remark: The feature is helpful.

10) **Export/Download**

- Instantly finds the option to download the graph in the top bar
- Question: Is there a possibility to download other formats?

11) **Other Remarks**

- Good: Highlighting (but too short)
- Suggestions: duplicate detection, grouping of similar values, graph visualization, reposition search option

Participant C (Novice II)

1) **Upload**
    - Chooses file from hard drive and uploads it
2) **Add**
    - Add environment is instantly detected
    - Remark: I do not understand the concept of triples.
    - Finds matching URIs via the auto-completion feature
    - Bug: Place holder values are not always deleted
3) **Delete**
    - Filters triples using the subject-column search
    - Bug: Triples that have once been deleted and then restored cannot be deleted again
    - Interprets the delete icon as such and successfully deletes triples
4) **Undo**
    - Finds undo button at the bottom right immediately
    - Reverts the last actions without problems
5) **Object Edit**
    - Looks for some button to edit objects
    - Detects that clicking on object cells makes them editable
    - Tries to double click
    - Successfully edits triples
6) **Subject Edit**
    - Instantly knows how to edit subjects
7) **Bulk Edit**
    - Bulk edit icon is detected but interpreted as an undo icon

- Tries to select multiple cells at once
- Remark: I tried to mark multiple cells like in Excel.
- Had to abort this task and be shown what the icon does

8) **Triple Import**

- Does not find import environment immediately
- Tries to press the Fetch Graphs button at first
- Then detects the Enter Import Keywords input fields
- Correctly inputs keywords and chooses a preset type
- Learn effect visible – knew instantly what to do on the second and third iteration
- Suggestion: Separate the import stronger from the add environment

9) **Literal-to-URI-Conversion**

- Easily adds new triples and uses the auto-completion function
- Doesn't immediately detect the Select URI dropdown menu in the object cells, but finds them shortly after
- Selects appropriate URIs to substitute the literal objects with

10) **Export/Download**

- Remark: I am looking for a button that says "Save" or "Download".
- Question: What does "parsing" mean?
- Suggestion: Change "Parse to RDF" to "Save" since "parsing" isn't a commonly used word

11) **Other Remarks**

- Good: Bulk edit, import features, conversion
- Remark: There is no text cursor when using the search fields.

Participant D (Expert II)

1) **Upload**

   - Chooses file from hard drive and uploads it
   - Suggestion: Upload the file automatically as soon as it has been selected

2) **Add**

   - Immediately found the add environment
   - Does not use the auto-completion feature at first, but then does during the second and third iteration
   - Rather uses mouse than arrow keys to navigate the auto-completion list
   - Suggestion: Give the add environment a heading

3) **Delete**

   - Uses multiple column searches to filter triples
   - Delete icon is recognized as such
   - Deletes triples successfully
   - Suggestion: Empty the search fields when a triple has been added or deleted

4) **Undo**

   - Uses the undo button at the bottom right corner
   - Successfully restores the previously deleted triples
   - Remark: I expected the button to be above the table.
   - Remark: If searches/filters are active some of the restored triples are not shown.

5) **Object Edit**

   - Uses table-wide search to filter triples
   - Tries to right-click object cells
   - Tries to double click object cells
   - Remark: I tried right-clicking an object cell and expected some kind of menu to pop up.

- Figures out how to edit object values shortly after and applies the edit correctly
- During the second and third iteration editing objects was easy

6) **Subject Edit**

- Edits subject successfully
- Remark: One could speed up the process of adding triples with simply pressing Enter key instead of having to push the add button.

7) **Bulk Edit**

- Bulk edit icon is detected but interpreted as an undo icon
- Had to abort this task and be shown what the icon does
- Suggestion: Show a tooltip to clarify what the icon does.

8) **Triple Import**

- Import environment is immediately found
- Correctly uses the Enter Import Keywords input field as well as the Choose Type button
- First presses the Fetch Graph button and then vice versa
- Successfully imports triples

9) **Literal-to-URI-conversion**

- Triples with literal values are being added without problems
- Remark: I expected the URI suggestions next to the Add button instead of in the object cell
- Still finds the Choose URI dropdown list as soon as the new triple was added
- Remark: The table loses its focus on the edited triple as soon as the conversion has taken place.

10) **Export/Download**

- Immediately finds the Parse to RDF button
- Suggestion: Use “Save” or “Download” instead of “Parse to RDF”.

- Remark: The icon makes it clear what the button does.

11) **Other Remarks**

- Good: Filters, performance, speed, bulk edit, literal-to-URI-conversion
- Suggestion: Selective bulk edit
- Suggestion: Use label-fields of graphs instead of their URIs to make them better human-readable.
- Suggestion: Tooltips for all actions and buttons
- Suggestion: Stronger separation of the search and add environment

F Enclosed DVD

The DVD enclosed to this thesis contains the following data:

- An electronic version of this thesis in the PDF format,
- a copy of the source code of *rdfedit* as it can be found on github.com with the commit-version November 8th, 2014[76]
- as well as the screencasts recorded for the thinking aloud tests conducted for this thesis.

[76] `https://github.com/suchmaske/rdfedit/commit/f8c2c2af316db76b16f9d8f03716f3da1b8b8e75`

G Eidesstattliche Erklärung

Hiermit versichere ich, dass ich die vorgelegte Arbeit eigenständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die Stellen der Arbeit, die im Wortlaut oder wesentlichen Inhalt aus anderen Werken oder dem Internet entnommen wurden, mit genauen Quellenangaben kenntlich gemacht habe. Verwendete Informationen aus dem Internet sind den Gutachtern gegebenenfalls auf Nachfrage vollständig zur Verfügung zu stellen. Die Arbeit hat in dieser oder ähnlicher Form noch nicht im Rahmen einer anderen Prüfung vorgelegen.

Berlin, den 10. November 2014 _____

Unterschrift